Rundquist

ARGUS

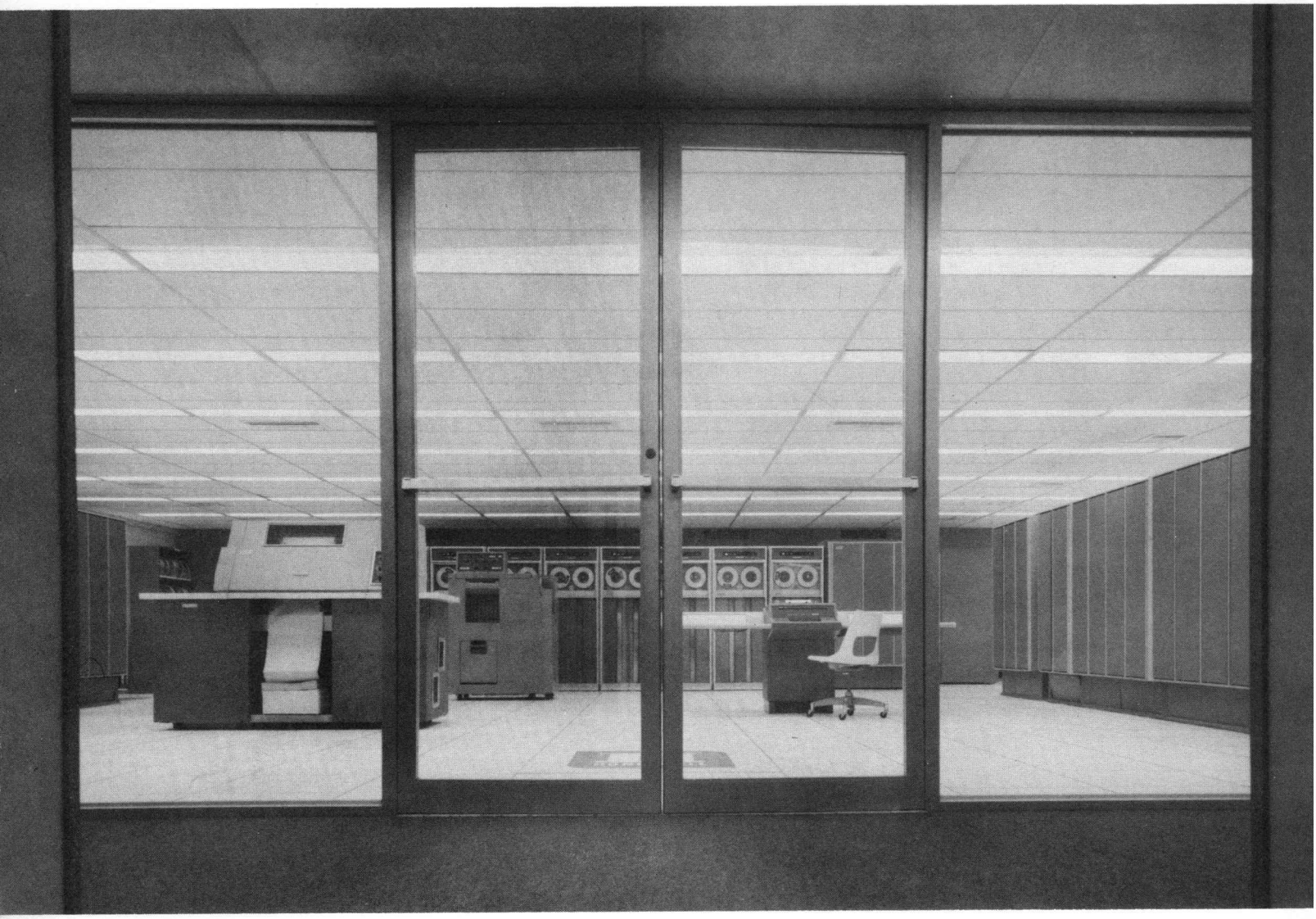
AUTOMATIC ROUTINE GENERATING AND UPDATING SYSTEM

MANUAL OF ASSEMBLY LANGUAGE

HONEYWELL

HONEYWELL 800

Transistorized Data Processing System



Litho in U.S.A.

DSI-23C
2461
2861

ARGUS

AUTOMATIC ROUTINE GENERATING AND UPDATING SYSTEM

MANUAL OF ASSEMBLY LANGUAGE

Honeywell

H HONEYWELL *Electronic Data Processing*

TABLE OF CONTENTS

TABLE OF CONTENTS (cont)

LIST OF ILLUSTRATIONS

SECTION I

INTRODUCTION

The ARGUS System

ARGUS, the Automatic Routine Generating and Updating System, is the core of the integrated automatic programming system for the Honeywell 800. ARGUS is designed to minimize programmer effort and to maximize the efficiency of every phase of program preparation, from the initial coding through the checkout phase to actual production. Wherever possible, the burden of routine, clerical operations is lifted from the programmer and the full power of the Honeywell 800 is brought to bear on such operations. The file-of-programs approach, whereby batches of programs are assembled, tested, modified, and scheduled for production, minimizes setup time by eliminating a great multiplicity of brief, repetitive computer runs. The dynamic dumping technique employed by the Program Test System enables batches of programs to be tested at full machine speed and without interruption. Diagnostic information is obtained without manual intervention, even if a programming error forces premature termination of a particular program under test. In short, ARGUS achieves a mating between the efficiency of program preparation and the remarkable efficiency of production made possible by Honeywell parallel processing.

As illustrated in Figure 1, ARGUS is composed of the following principal elements:

1. An Assembly Program which translates symbolic coding and produces operating programs in machine language (binary) on magnetic tape;
2. A Library of Routines containing both subroutines and macro routines, each thoroughly tested and capable of being incorporated into any program during assembly by the inclusion of a single pseudo instruction;
3. A Library Additions and Maintenance Program (LAMP) for adding and deleting routines and modifying existing routines in the library;
4. A Program Test System which operates a file of unchecked programs at full machine speed, automatically obtaining requested information at points specified by the programmer for later analysis of program operation;
5. An Executive System which schedules checked-out programs for parallel processing, based on their individual hardware requirements, timing, and urgency, and then automatically loads and executes the scheduled programs.

A program to be processed on the Honeywell 800 may be prepared in ARGUS assembly language, as described in this manual, or it may be written in the language of either the Algebraic or FACT (Business) Compiler and automatically converted to assembly language. In either case, the Assembly Program translates this language and produces an operating

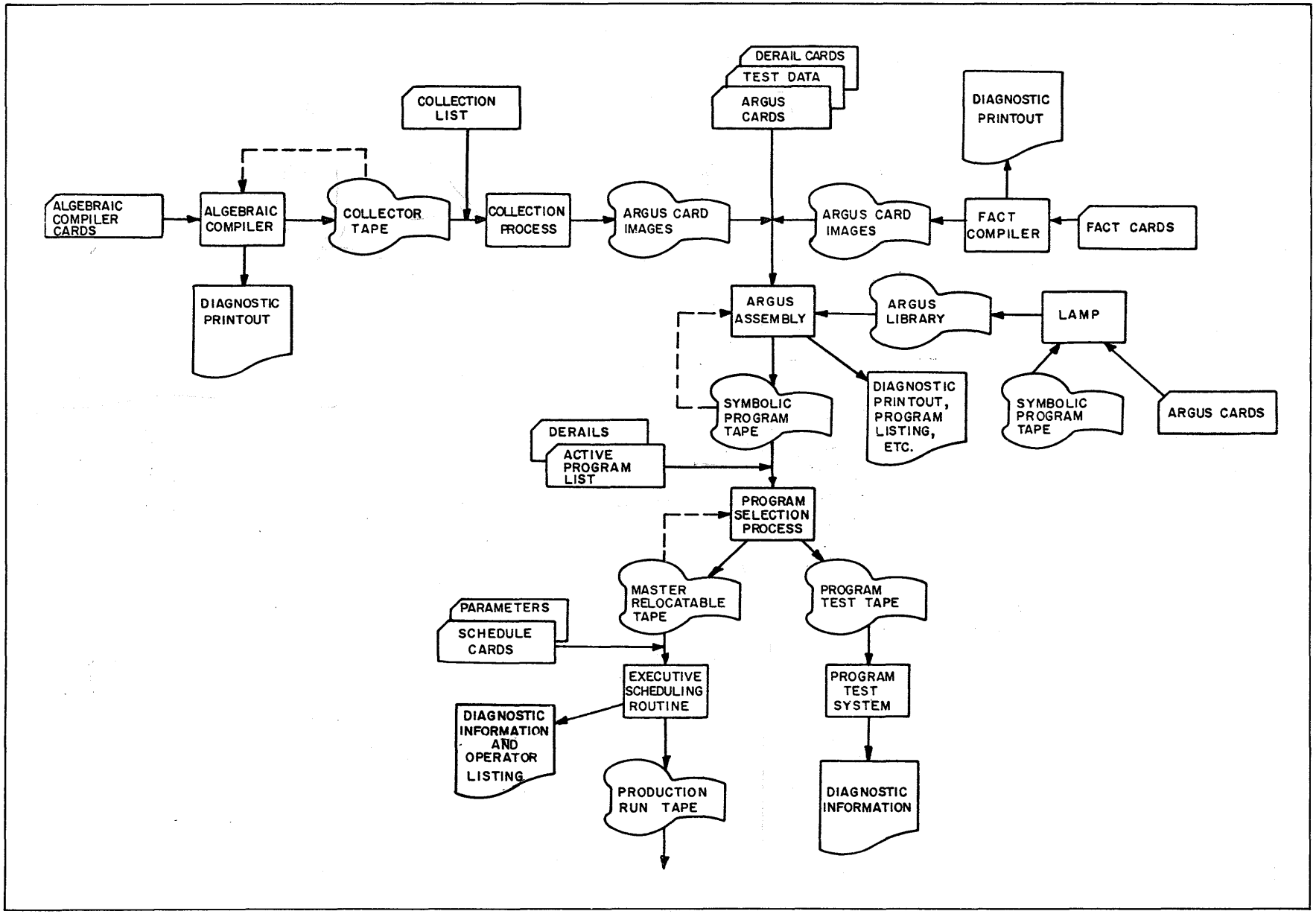


Figure 1. Honeywell 800 Automatic Programming System

program in both symbolic and machine language on the symbolic program tape, which contains a file of programs being checked out, together with test data for each program. This is accomplished as part of an updating run in which programs are added to or deleted from the symbolic program tape, and existing programs and test data are modified on the basis of information derived from the preceding checkout run.

The updating run is normally followed by a program selection run which can prepare a program test tape, containing programs and test data to be executed during a checkout run, or a master relocatable tape, containing checked-out programs to be scheduled by Executive for production operation, or both. Input to this run includes the symbolic program tape, plus an active program list which specifies the programs to be transferred to either of the output tapes. Those programs transferred to the program test tape are accompanied by test data and by derail instructions which specify the kinds and amounts of diagnostic information to be generated during checkout and the program points where this information is to be generated. The program test tape is the input to the checkout run, during which the ARGUS Program Test System executes each program on the tape, using the accompanying test data and generating the requested information at the specified points. The information generated is printed in a variety of formats designed for convenient analysis by the programmer, who uses this information to specify the changes in programs and test data which will be effected during the next updating run.

Programs transferred to the master relocatable tape are accompanied by relocation information which Executive uses to modify memory and peripheral equipment assignments in order to schedule production programs for automatic parallel processing. The master relocatable tape is the input to the Executive scheduling run, together with a proposed schedule which specifies the programs to be scheduled, the memory and equipment requirements of each, and any necessary production sequence among the various programs. Based on this information, Executive schedules groups of programs to be processed in parallel, relocates the scheduled programs, and records them in operating form on the production run tape.

The Executive run supervisor is also stored on the production run tape, along with the scheduled programs. This routine executes the schedule, automatically loading the production programs, turning them on and off, and communicating with the operator as necessary. Manual intervention during the production run is minimized, but may be used to alter the schedule being performed or to handle any unexpected occurrences.

The Assembly Program

As noted above, the Assembly Program translates coding written with mnemonic and symbolic codes and produces operating programs in machine language. Programs written in assembly language are independent of fixed memory locations and may be modified, corrected, or expanded in assembly language by the computer. The ARGUS Assembly Program offers the following automatic programming features: symbolic and relative reference, allocation, translation, library routine insertion, sort generation, and relocation.

SYMBOLIC AND RELATIVE REFERENCE: Since programs are written without reference to fixed memory locations, program words may be referred to by means of symbolic tags. However, it is not necessary to tag all of the words of a program. Untagged words may be referred to relatively, using address arithmetic to specify the desired location either:

1. Relative to the location of a tagged word; or
2. Relative to the location of the word containing the reference.

ALLOCATION: Program words are automatically allocated in the high-speed memory according to the sequence in which they are assembled. The programmer may specify the location of the first word in any sequence, if he so desires. Although the allocation of memory locations normally proceeds automatically, flexibility is enhanced by the provisions for programmer control of this process.

TRANSLATION: ARGUS instructions are written using mnemonic operation codes, symbolic or relative addresses, and decimal numbers. The Assembly Program translates these into the binary language of the Honeywell 800. Constants written in alphabetic, decimal, octal, or mixed form are translated into binary-coded alphanumeric, fixed-point or floating-point binary-coded decimal, or fixed-point or floating-point binary form.

LIBRARY ROUTINE INSERTION: A library of useful, thoroughly tested subroutines and macro routines is readily available to each Honeywell 800 installation, so that frequently used coding is preserved for easy insertion into new programs. Each subroutine or macro routine in the library is represented by a pseudo instruction which specifies the desired routine plus all parameters required for its execution. These pseudo instructions may be included in a program as easily as machine instructions. When they are processed, ARGUS obtains the corresponding coding from the library and inserts it into the program.

SORT GENERATION: Included in the Library of Routines is a group of sort generators which can produce routines tailored to specific sorting applications. The programmer includes in

his program a pseudo instruction which specifies the type of sort desired and the equipment available for its execution. The description of the format of the data to be sorted is included with the data itself. ARGUS sort routines are composed of two phases: a presort phase which produces ordered strings of data and a merge sort phase which combines these ordered strings to form a single over-all sequence. A new and unique method of merging is used which takes optimum advantage of any available number of magnetic tape units.

RELOCATION: ARGUS retains a record of the structure of each program word so that any assembled program may be automatically relocated to operate in another portion of the high-speed memory or to utilize other special registers, magnetic tape units, or input/output terminal units. This feature of ARGUS greatly facilitates the use of the parallel processing ability of the Honeywell 800.

SECTION II

THE HONEYWELL 800

Word Structure

Information is handled by the Honeywell 800 in fixed-length words comprising 54 binary digits, or bits. Six of these bits are reserved for the automatic checking circuits and may be disregarded by the programmer. The 48 information bits of each word may be grouped to form several basic types of words as shown in Figure 2.

In an instruction, the information bits are divided into four 12-bit groups which represent the command code and the A, B, and C addresses, respectively. The address groups normally designate the locations of operands, but in certain instructions they contain other special information. The command code group contains, in addition to the operation code, other special information which depends upon the type of instruction. The complement of Honeywell 800 machine instructions is presented in Appendix E.

In a fixed-point constant, the 48 information bits may represent eight alphanumeric characters, 11 signed or 12 unsigned decimal or hexadecimal digits, 15 signed or 16 unsigned octal digits, 44 signed or 48 unsigned binary digits, or any combination of characters, digits, and bits not exceeding 48 bits. Up to four individually signed fixed-point constants, having an aggregate of not over 44 bits, may be stored in a single word. A floating-point constant may consist of a seven-bit exponent and a 40-bit mantissa, with sign, or a seven-bit exponent and a 10-decimal-digit mantissa, with sign. Section IX describes the specification of data constants in assembly language.

Information Storage

The Honeywell 800 main (or high-speed) memory is composed of banks, each capable of storing 2048 machine words. These memory banks are available in pairs called modules, and a system may include from one to four such modules (4096, 8192, 12,288, or 16,384 words). Every main memory location is designated by a unique address, consisting of a bank indicator from 0 to 7 and a subaddress from 0000 to 2047.

In addition, the Honeywell 800 contains a special control memory which selects instructions and operand addresses. The control memory is fixed in size and consists of eight identical groups of 32 special registers each. A special register is designated by a unique

address, consisting of a group indicator from 0 to 7 and a subaddress from 00 to 31. Each special register has the capacity to store a complete main memory address (sign, bank indicator, and subaddress).

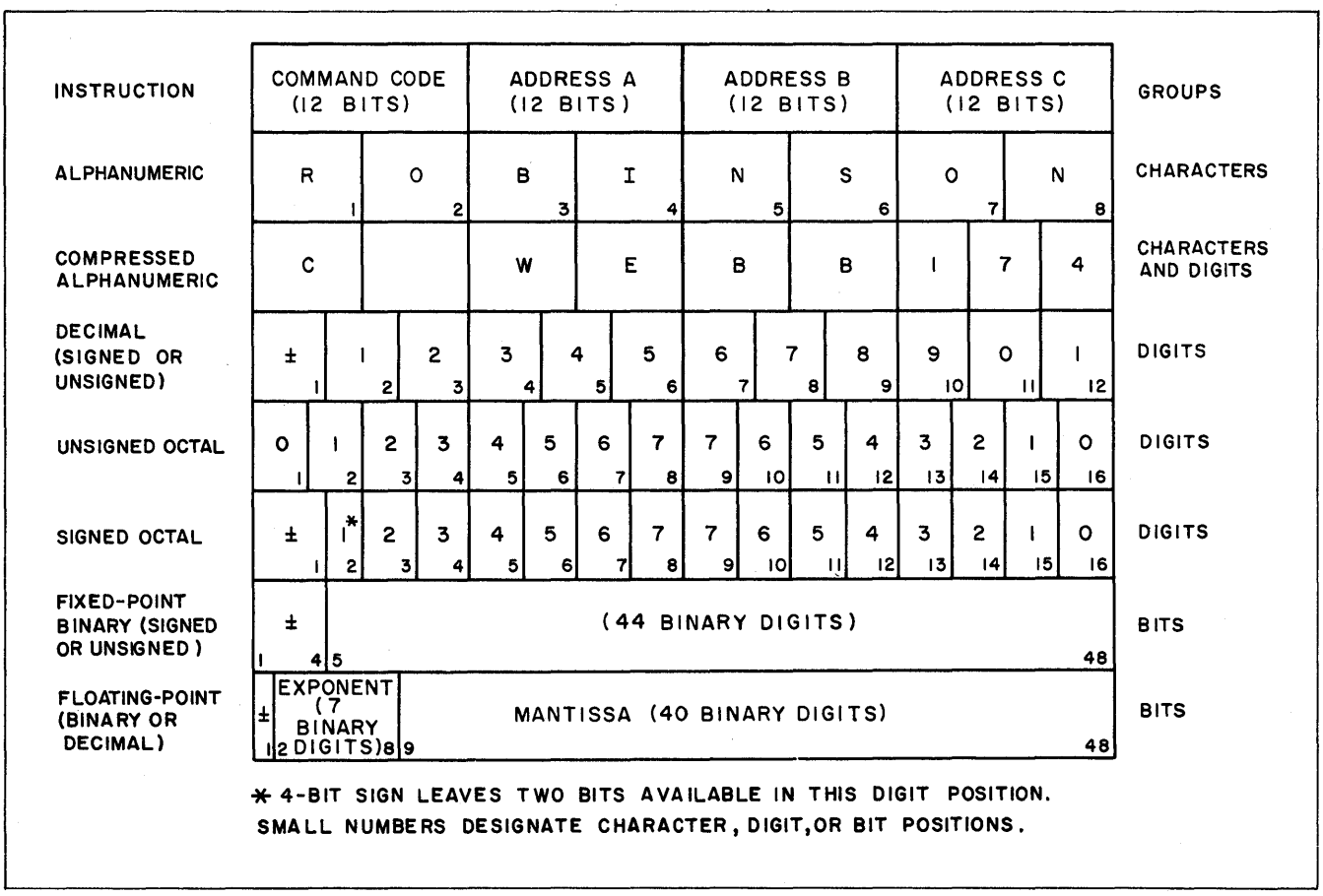


Figure 2. Honeywell 800 Word Formats

Sequence Control

The operational control of an individual program is delegated to a specific special register group. Each group includes a pair of functionally identical sequencing counters, called the sequence counter (SC) and the cosequence counter (CSC). Whenever one of these counters selects an instruction for execution, the contents of the counter are automatically incremented by 1. Most machine instructions have the ability to designate one of these counters as the source of the next instruction. Those instructions which do not include this facility are followed by an instruction selected by the same counter. Instructions which result in a programmed change of sequence always alter the contents of the counter designated as the source of the next instruction.

Also included in each special register group is a pair of history registers, called the sequence history register (SH) and the cosequence history register (CSH). Whenever the contents of a sequencing counter are altered, other than by normal incrementing or direct addressing, the corresponding history register stores the incremented contents of the counter which produced the sequence change. Thus it is possible to depart from a programming sequence and execute several instructions under control of the alternate counter before returning to the original sequence, or to program a sequence change and automatically retain a record of the next step that would have been performed had the change not occurred.

Command Codes

The command code group in a Honeywell 800 machine instruction contains an operation code which specifies the instruction to be performed. Depending upon the type of instruction specified, this group may also contain such information as a peripheral code, a partial mask address, memory designator bits which relate each of the three address groups to either main or control memory, and a bisequence bit which indicates the source of the following instruction.

Machine instructions are of five types: general, shift, peripheral, simulator, and scientific, as distinguished by the various command code formats. The details of these formats are described in Section VII. General instructions include arithmetic operations, information transfers, decisions, and other familiar data processing functions. Many of these instructions can manipulate variable-length fields by the use of masks. These are called field instructions. Shift instructions are always performed with masked operands. Peripheral instructions perform all operations which involve magnetic tape units and terminal input/output equipment, such as reading, writing, and rewinding. Simulator instructions are defined by the programmer to represent, by means of a single instruction, an entire body of coding. Scientific instructions manipulate data in floating-point form, which greatly improves the efficiency of scientific computations.

Addresses

Every memory location and every special register in the Honeywell 800 has a unique numerical designation or address. An instruction may refer to any memory location to obtain an operand or to store a result. Unmasked general instructions and all shift instructions may refer to special registers, using memory designator bits to denote this type of addressing. Masked general instructions do not have this ability.

A direct address is an explicit statement of the address of the desired operand. An indexed address is written by specifying an index register and a quantity which augments the contents of this register to form the desired address. (Eight index registers are included in each special register group.) An indirect address is written by specifying a special register in which the desired address is stored, plus an increment which permanently alters the special register contents after use. Six types of addresses may be written in the address groups of instructions. A memory location or a special register may be addressed directly. The augmented contents of an index register may be interpreted as a memory location address or as a special register address. A memory location may be addressed indirectly by referring to a special register where the desired address is stored. Finally, the special register used to obtain an indirect address may be specified by indexed addressing. The ARGUS formats of these addressing options are presented in Section V.

The specifications of certain machine instructions direct that one or more address groups contain information other than references to memory locations or special registers. Such information may include, for example, the number of words to be transferred, the number of positions to shift an operand, or the partial designation of a mask location.

Masks

Reference to a word called a mask in an instruction permits the designation of partial words as operands and as a result. The mask designates the character, digit, or bit positions within the operand words on which the stated operation is to be performed. With certain of the general instructions (field instructions) the use of a mask is optional, but with shift instructions a mask is always required.

When an arithmetic operation is masked, the mask is applied to both operands and to the result. Shift masks are applied after shifting and before delivery to the result location. All masking in the Honeywell 800 leaves the unmasked portions of the result location unchanged (protected masking) except for the extract instruction (EX) and the two shift and extract instructions (SWE and SPE), which clear the unmasked portions of the result location to binary zeros (unprotected masking).

Masks are stored in the high-speed memory in groups of consecutive locations. An instruction which uses a mask references the desired mask relative to an address known as the base of the mask group. Each special register group includes a mask index register (MXR) which stores two mask group bases, according to a specified format: the base of a group of field masks and the base of a group of shift masks (see Section X). The relative

position of the desired mask within the group is specified in the instruction using the mask. The base of a field mask group must be a multiple of 32 and the group includes up to 32 masks. A field instruction can specify in its command code field any of the masks in this group. The base of a shift mask group must be a multiple of 64 and the group includes up to 64 masks. A shift instruction can specify in the B address field any of the masks in this group. Therefore, each setting of the mask index register makes 96 memory locations available to the programmer for the storage of masks. To conserve memory space by making certain masks available for use with either type of instruction, the two mask groups can be made to overlap. With such an MXR setting, 64 memory locations are made available for the storage of masks, of which 32 can be used with field instructions and all 64 with shift instructions.

The programmer has the option of specifying the address of the desired mask in memory, or including information in the command code field which enables ARGUS to generate the desired mask (with the exception of the substitute (SS) and extract (EX) instructions, which always require a programmer-specified mask).

SECTION III

THE ARGUS CODING FORM

Programs to be assembled by ARGUS are written using the coding form shown in Figure 3. The coding on these forms is then punched on standard 80-column cards according to the fixed-field format shown in Figure 4. An instruction word occupies an entire line on the coding sheet and an entire punched card. Constants may be combined, however, to allow punching of more than one on a single card. When an entire program deck, complete with all necessary control instructions, is assembled by ARGUS, the program is produced in operating form on magnetic tape. In addition, ARGUS produces a listing of the program in printed form. Assembly outputs are described in Section XII.

Figure 4 shows that the ARGUS input card contains seven fixed fields. The function of each of these fields is described briefly in the following paragraphs. The format of each input word type is illustrated in detail in the following sections.

Location Field (Columns 1-10)

The location field may contain tags symbolizing memory locations, special registers, or masks. Any word which is to be referred to symbolically in an address field of some other word, or which is to be loaded into a special register, or which is to be used as a mask, must include a tag in this field. Tags may be punched anywhere in the location field; spaces are ignored. The types and formats of tags are described in Section IV.

Command Code (Columns 11-23)

The command code field is divided into two subfields. Columns 11 through 22 contain the command code group itself, while column 23 designates the source of the next instruction. If column 23 contains an "S" or if this column is blank, the next instruction will be taken from the sequence counter in the assigned special register group. If column 23 contains a "C", the next instruction will be taken from the cosequence counter in this group. This column is not used in a peripheral or a simulator instruction or in the instruction proceed (PR).

Columns 11 through 22 may contain the mnemonic operation code of a machine instruction, followed by any other information required by that instruction, as described in Section VII. In the case of a constant or set of constants, the command code field contains the constant code and any other required information, as described in Section IX. This field may also be

Honeywell
Electronic Data Processing

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	REMARKS 66 LINE NUMBER 73	74 80
1								
2								
3								
4								
5								
6								
7								
8								
9								
10								
11								
12								
13								
14								
15								
16								
17								
18								
19								
20								

Figure 3. The ARGUS Coding Form

used to specify an ARGUS control instruction or the pseudo instruction of a library routine. The formats of these words are described in Sections VIII and XIII, respectively.

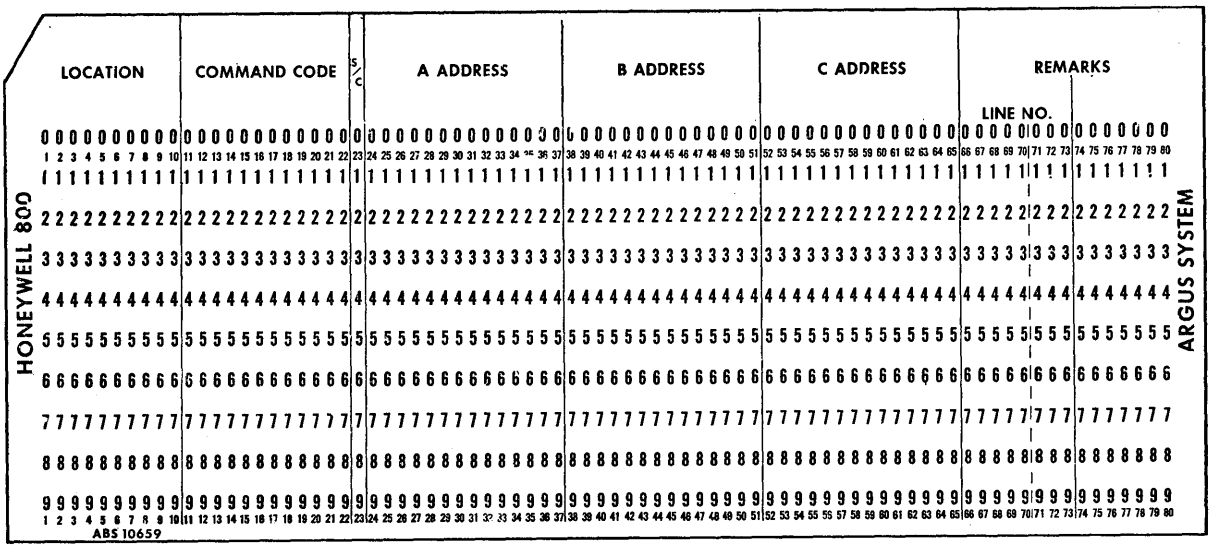


Figure 4. The ARGUS Input Card

Address Fields (Columns 24-37, Columns 38-51, Columns 52-65)

These three address fields correspond to the A, B, and C address groups of a Honeywell 800 machine instruction. In a machine instruction, they may designate an operand location or a result location, using any of the six types of addresses permitted by the instruction. These six address types are described in Section V. In certain instructions, the address fields may contain instruction parameters or other information. The three address fields are regarded as a single 42-column field for the purpose of punching constant words. Their format in a library routine pseudo instruction is determined by the programmer who designs the library routine.

Line Number (Columns 66-73)

Line numbers specify the sequence of words within a program. When a new program is assembled, the cards may or may not contain line numbers. If the cards do not contain line numbers, they must be read in correct sequence, as ARGUS assigns a line number to each card based on this sequence. If the cards contain line numbers, ARGUS sorts the cards into proper sequence.

Line numbers are printed as part of the complete program listing produced by ARGUS. They are used by the programmer in preparing additions, deletions, and corrections to assembled programs. Five-digit line numbers are originally assigned the cards of a new program. If assigned by the programmer, they are punched in columns 66 through 70. To

correct or replace one of the original cards of a program, the assigned five-digit number is punched in columns 66 through 70 of the modification card. Columns 71 through 73 are used to insert additional cards in the correct sequence. For example, if three cards are to be inserted in a program following card 01357 (according to the program listing), line numbers may be punched on the inserted cards as follows:

Column	66	67	68	69	70	71	72	73
Original Card	0	1	3	5	7	0	0	0
1st Insert	0	1	3	5	7	1	0	0
2nd Insert	0	1	3	5	7	2	0	0
3rd Insert	0	1	3	5	7	3	0	0
Original Card	0	1	3	5	8	0	0	0

Again at some later time, the programmer may insert additional cards following card 01357200 by numbering them 01357210, 01357220, etc.

Identification (Columns 74-80)

These columns may contain a punch combination used to identify the cards of a related set of coding, such as a program segment. If such codes are used as segment markers, for example, ARGUS can identify the segment to which each program card pertains.

Remarks (Columns 66-80)

If either the line number or identification field (or both) is not so used by the programmer, it may contain remarks. Such information is not assembled but is reproduced for the programmer's convenience as part of the program listing.

A card containing only remarks may be included at any point in a program. Such a card is indicated by an "R" or a "P" followed by a comma in columns 1 and 2 of the location field. (R causes the remarks to be printed on the next line; P causes the remarks to be printed at the top of the next page.) Remarks may be punched in all of the other columns (3-80) of a remarks card.

SECTION IV

TAGS

A tag punched in the location field of a program word allows the programmer to refer to that word elsewhere in his program without being aware of its absolute location in memory. A word may also be tagged to denote its use as a mask or to direct its storage in a special register. Three groups of words must include a tag punched in the location field:

1. Certain of the words which are directly referenced in the address fields of other program words;
2. All words which are to be placed in special registers at loading time; and
3. All masks.

Tags may be punched anywhere in the location field; spaces are ignored.

Symbolic Tags

A symbolic tag is a group of up to eight alphanumeric characters, of which at least one must be non-numeric. However, there are certain characters which have significance to ARGUS and must not be included in symbolic tags.

Key Punch	Symbol	Machine Code
12	+ (plus)	010000
11	- (minus)	100000
0, 8, 3	, (comma)	111011
12, 8, 3	. (period)	011011
11, 8, 4	* (asterisk)	101100
0, 1	/ (slash)	110001

In addition, the following characters are not permitted within symbols even though they have no special significance.

Key Punch	Symbol	Machine Code
8, 4	- (hyphen)	001100
11, 8, 5	" (quotes)	101101
12, 8, 2	; (semicolon)	011010
8, 5	(not assigned)	110000
0, 8, 7	(not assigned)	111111

Space codes (001101) within symbolic tags are ignored by assembly.

Tags are frequently chosen as mnemonic representations of the content or function of the tagged words, e.g., GROSSPAY, INPUT1, or DIVIDEND. Such a tag may directly

represent any location in the high-speed memory. Every symbolic tag which appears in an address field within a program must appear in the location field within that program.

It is not necessary to tag every word of a program which is referenced by some other word. Address arithmetic (described in Section V) allows direct reference to an untagged word by specifying its location relative to a tagged word, e.g., GROSSPAY + 2. The programmer decides which words of his program to tag and which to reference by address arithmetic. Note that for purposes of assembly, address arithmetic is permitted only in an address field and never in the location field. [1]

Normally every symbolic tag appearing in the location field is assigned an absolute value by ARGUS. The program listing includes the assignment of each tag. These assignments are used if the program is loaded independently, as is usually the case during program testing. However, in production the program is generally loaded under the direction of Executive and the tag assignments are thereby modified to make the program compatible with any other programs being processed in parallel.

In addition to their use in referencing program words directly, symbolic tags may be used to represent other values, such as complete addresses in indexed or indirect form, or program parameters. The programmer assigns the values of such tags using special ARGUS control instructions provided for this purpose. These instructions, called EQUALS, ASSIGN, and TAS (temporary assign), are described in Section VIII.

Special Register Tags

Each of the 32 special registers in a group has both an absolute address and a mnemonic designation. The names and the absolute and mnemonic addresses of all special registers in a group are listed in Figure 5. For example, this figure shows that the mask index register in any group may be designated absolutely as 07 or mnemonically as MXR.

A special register tag is required in the location field of every word to be loaded directly into a special register. Such a tag consists of a "Z" followed by a comma and the absolute or mnemonic address of the desired register. For example, either of the following special register tags

[1] PTS derail instructions are an exception to this rule, as described in the Program Test System Manual, Section III, "Expressing Memory Addresses".

Z, 11
Z, X3

might be used to load the tagged word into index register 3. For a discussion of special register tags used in address fields, see Section V.

Subaddress	Mnemonic Address	Name
00	AU1	AU-CU Counter No. 1
01	AU2	AU-CU Counter No. 2
02	SC	Sequence Counter
03	CSC	Cosequence Counter
04	SH	Sequence History Register
05	CSH	Cosequence History Register
06	UTR	Unprogrammed Transfer Register
07	MXR	Mask Index Register
08-15	X0-X7	Index Registers
16-23	R0-R7	General Purpose Registers
24-31	S0-S7*	General Purpose Registers
28	RAC	Read Address Counter
29	DRAC	Distributed Read Address Counter
30	WAC	Write Address Counter
31	DWAC	Distributed Write Address Counter

*In certain special register groups, S4-S7 are replaced by RAC, DRAC, WAC, and DWAC.

Figure 5. Special Register Names, Subaddresses, and Mnemonic Addresses

Mask Tags

Every mask specified by the programmer must be designated in the location field by a unique symbolic tag. These tags, like all symbolic tags used with ARGUS, can have up to eight alphanumeric characters, of which at least one must be non-numeric. In addition, each such tag is preceded by a character which indicates that the corresponding mask is used with field instructions (F), shift instructions (S), or both (B). Thus, a complete mask tag consists of a mask indicator followed by a comma and a symbolic tag.

F, M3
S, RIGHT2
B, SIGN

Link Tags

Any word which is to be the starting location of a segment (except the starting location of the first segment) should be so marked by tagging the word with a symbolic tag preceded by the letter "L" and a comma.

Out-of-Sequence Words

It is sometimes convenient, particularly when writing macro routines, to have certain words placed out of the main sequence of coding. ARGUS recognizes any word marked by the letter "X" and a comma in the location field as an out-of-sequence word. Such words are placed at the end of the subsegment in which they appear. The "X, " may or may not be followed by a symbolic tag.

ARGUS assigns out-of-sequence words by maintaining two location counters called CLC (current location counter) and XLC (out-of-sequence location counter). Each counter is incremented after a word of the corresponding type is processed. A word without "X, " in the location field is assigned to the location contained in the CLC. A word with "X, " in the location field is assigned to the location contained in the XLC.

Definition of Tags

When a tag appears in the location field of a line of coding, it becomes defined. This results in the assignment of the tag to a memory location, an integer, or a complex address (i. e., an indexed address or a special register address). A tag may have one absolute assignment (memory location or integer) or one complex assignment or one of each. However, when a tag has conflicting assignments (e. g., two memory location assignments), it becomes doubly defined and is noted by ARGUS as an error. In general, such a conflict of assignment can arise only within a single segment. In other words, a tag may have completely different assignments in the various segments of a program. The only tags which must maintain their assignments throughout the entire program are link tags and tags which appear within the common portion of any segment (see Section VI).

When a tag which has both an absolute assignment and a complex assignment appears in an address field, the complex assignment is normally used. However, there are several exceptions to this rule, which are noted in connection with machine instructions, control instructions, and control constants.

SECTION V
ADDRESSES

In Section II, it was stated that every Honeywell 800 main memory location has a unique numerical designation, or address, consisting of a bank number from 0 to 7 and a subaddress from 0000 to 2047. It was stated further that each control memory location, or special register, is uniquely designated by a group number from 0 to 7 and a subaddress from 00 to 31.

Most instructions can refer to any memory location or special register to obtain an operand or to store a result. Three methods of addressing main and control memory are provided. A direct address is a specific reference to the desired location or register. An indexed address designates a special register called an index register, plus a quantity which augments the contents of the index register to form the desired address. This process leaves the original contents of the index register unaltered. An indirect address designates a special register in which the desired address is stored, plus an increment which permanently modifies the stored address after use. The internal configurations of the various types of addresses are presented in the Programmers' Reference Manual. This section deals with their representation in ARGUS language, as summarized in Figure 6.

Direct Memory Location Address

The programmer may directly reference a memory location by writing the symbolic tag assigned to that location in an address field. ARGUS replaces this tag with the absolute address assigned. Alternatively, the programmer may specify a direct memory location address by means of address arithmetic (see below). Address arithmetic permits addressing relative to a tagged location or relative to one of the location counters (CLC and XLC) mentioned in Section IV.

Direct addressing may be used in an instruction to reference any location in the memory bank in which the instruction is stored. An attempt to address any location outside of this bank results in an ARGUS error indication during assembly. Therefore, the use of direct addressing is limited by the rules which govern relocation (see Section VI).

Type	Format	Interpretation
Direct Memory Location	(1) Symbolic Tag (2) Symbolic Tag, ± Address Modifier (3) C, ± Address Modifier (4) X, ± Address Modifier	Direct reference to a high-speed memory location in assigned bank (1) symbolically, (2) relative to a tagged location, (3) relative to the contents of the current location counter, or (4) relative to the contents of the out-of-sequence location counter.
Direct Special Register	(1) Z, 15, increment (2) Z, R2, increment	Direct reference to a related special register (1) absolutely, or (2) mnemonically.
Indexed Memory Location	(1) IR, numeric augmenter (2) IR, symbolic augmenter	Reference to a high-speed memory location in any bank formed by combining contents of a related index register with (1) numeric augmenter, or (2) symbolic augmenter (with or without modifier).
Indexed Special Register	(1) IR, Z, 15, increment (2) IR, Z, R2, increment	Reference to a special register in any group formed by combining the contents of a related index register with (1) an absolute special register subaddress, or (2) a mnemonic special register designator.
Indirect Memory Location	(1) N, 15, increment (2) N, R2, increment	Reference to a related special register addressed (1) absolutely, or (2) mnemonically to obtain stored absolute address of a memory location in any bank.
Indexed Indirect Memory Location	(1) IR, N, 15, increment (2) IR, N, R2, increment	Reference to a special register in any group formed by combining the contents of a related index register with (1) an absolute special register subaddress, or (2) a mnemonic special register designator to obtain stored absolute address of a memory location in any bank.

Figure 6. Summary of Addresses

ADDRESS ARITHMETIC: An address modifier, consisting of a sign and a number from 0 to 2047, may be appended to a symbolic tag to designate a direct memory location address relative to the location specified by the tag. Such an address modifier may be appended to a "C" and a comma (C,) to designate a direct memory location address relative to the contents of the current location counter, or to an "X" and a comma to designate a direct memory location address relative to the contents of the out-of-sequence location counter. Thus the address

ASSETS +37

is a direct reference to the memory location 37 beyond that represented by the symbolic tag ASSETS. The address

C, -3

refers to the memory location three before the location whose address is stored in the current location counter. Likewise, the address

X, +109

refers to the memory location 109 beyond the location whose address is stored in the out-of-sequence location counter. The address modifier may be a series of numbers separated by the signs + and -, provided that the absolute value of the entire modifier does not exceed 2047. Caution is required in the use of address arithmetic, since the address modifiers are not corrected if coding is inserted or deleted later.

Three types of direct memory location addresses are illustrated in the instruction

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

1 LOCATION 10	11 COMMAND CODE 22	S/C 24	A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	66 REMARKS LINE NUMBER 73 74 80
	DA		C, +2	INTEREST	AMTPAID-10	

The function of this instruction is to add decimally the contents of the memory location two after the location of the instruction itself to the contents of the memory location designated by the tag INTEREST, and to store the result in the location 10 before that tagged AMTPAID. (Since this instruction is not marked by an "X, " in the location field, the CLC contains the address of this instruction while the instruction is being processed.)

The number of symbolic tags required to write a program can be greatly reduced by the use of address arithmetic. The programmer decides how many and which words in a program to tag and which to reference by address arithmetic.

Direct Special Register Address

The direct address of a special register is indicated by a "Z", a special register designation, and an unsigned increment from 0 to 31, all separated by commas. The special register designation may be either the absolute subaddress (from 0 to 31) or the mnemonic address (e.g., X3 or MXR) of the desired register, as shown in Figure 5, page 19. If a special register is addressed as an operand location, the numeric increment is added, under control of the special register sign, to the special register contents, after those contents have been used. If a special register is addressed as a result location, the increment is ignored. To address a special register as an operand location without changing its contents, the programmer may omit the increment or may write an increment of 0.

Any directly addressed special register is defined as being in the 32-register group controlling the program. For example,

Z, X2, 5

is the direct address of the second index register in the controlling special register group. If it is used to specify an operand location, this address directs that the contents of X2 are to be incremented by 5 after use.

Indexed Memory Location Address

A special register group includes eight index registers, each capable of storing a sign, a bank indicator, and a memory location subaddress. An indexed memory location address designates an index register and a quantity which augments its contents to form a complete memory location address. The index register designator and the augmenter are separated by a comma. The index register designator is a number from 0 to 7 which specifies one of the eight index registers in the controlling special register group. Use of the letter "X" before the designator is optional. The augmenter may be a number from 0 to 255 (254 for index register 7) or it may be a symbolic tag, with or without a modifier. If symbolic, it must be assigned by an EQUALS instruction (see Section VIII) to a number which is a valid augmenter. The computer forms a memory location address by adding the augmenter to the address stored in the index register, under control of the stored sign. The unaugmented address is retained in the index register.

For example, the address

3, 15 (or X3, 15)

specifies that the contents of index register 3 in the controlling group are augmented by 15 to form an absolute memory location address. The address

7, DIVIDEND +2

specifies that the contents of index register 7 are augmented by 2 plus the quantity equated to the tag DIVIDEND to form the desired address. If the sum of the augmenter plus the stored subaddress exceeds 2047, a carry occurs into the bank indicator and the resulting address will be in a different bank from the stored address.

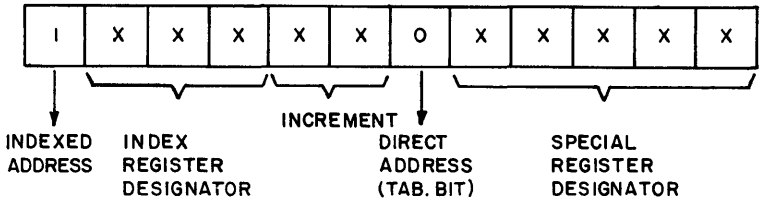
Indexed addressing permits the programmer to address locations in any main memory bank, depending upon the value of the bank indicator stored in the index register. This type of addressing may be used in processing multi-word items or in referring to a stored table. The address of the first word in the item or table is stored in an index register and all references to the item or table are made using the index register designator with the appropriate augmenter. To assure positive augmentation, the programmer must take care that the index register contains a positive sign.

Indexed Special Register Address

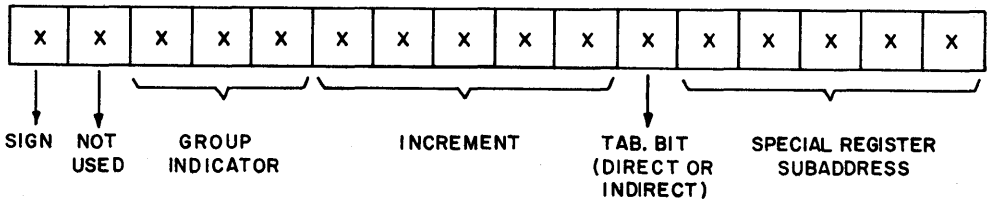
An indexed special register address may be used to refer to a special register in any of the eight control memory groups. Such an address takes the form:

Index Register Designator, Z, Special Register Designator, Increment

The index register designator is a number from 0 to 7 (or X0 to X7) which specifies one of the eight index registers in the controlling group. The special register designator may be an absolute subaddress (0-31) or it may be mnemonic (see Figure 5). The increment may be a number from 0 to 3 or it may be omitted. The manner in which these numbers are used to augment the index register contents and form a special register address is illustrated in terms of bit structure. ARGUS converts the address as written by the programmer to the following 12-bit configuration:



Of this configuration, the low-order eight bits (increment, tabular bit, and special register designator) are added to the low-order eight bits of the index register contents, under control of the index register sign, permitting carry into the high-order bits. As usual, the indexing process does not alter the contents of the index register. The augmented index register contents are interpreted by the machine as a special register address, as follows:



Within the augmented configuration, the group indicator and subaddress uniquely define a special register in any of the eight groups. The increment is now a number from 0 to 31. The tabular bit indicates whether the type of addressing is direct or indirect (see below). In either case, the increment, under control of the special register sign, is added to the contents of the special register after use, provided the special register is not addressed as a result location.

If the index register used contains all zeros (except for the sign and the group indicator), the result of indexed special register addressing is quite simple. In this case, the index register designates the group and the programmer designates the subaddress of a special register which is addressed directly and incremented after use by the amount which the programmer writes. For example, assuming that the programmer writes

3, Z, 5, 2

and that index register 3 contains

+4, 0, 0, 0

(group indicator of 4, increment, tabular bit, and special register subaddress of all zeros), the machine addresses special register 5 in group 4 directly and then increments its contents by +2. However, if the index register contains more than a sign and a group indicator, the result of indexed special register addressing can only be understood by combining bit configurations as above.

Indirect Memory Location Address

This address takes the form

N, Special Register Designator, Increment

where the special register designator specifies a register in the controlling group absolutely or mnemonically, and the increment is a number from 0 to 31. The machine interprets the contents of the specified register as the bank indicator and subaddress of a memory location which may be in any bank. Whether the memory location is an operand or a result location, the increment is added to the contents of the special register, under control of the special register sign, after they have been used. For example, the address

N, R3, 9

specifies the contents of special register R3 in the controlling group, interpreted as an absolute

memory location address. After use, the contents of register R3 are permanently modified by 9.

Indirect addressing is convenient for processing multi-item records when an operation is to be performed on word M of each item. The location of word M of the first item is stored in a special register. This location is then addressed indirectly, using an increment chosen to reset the special register to the location of word M of the second item. Since the bank indicator of the memory location is derived from the special register, any memory bank may be addressed in this fashion.

Indexed Indirect Memory Location Address

As noted in the discussion of indexed special register addressing, the augmented contents of an index register may be interpreted as a special register group indicator and subaddress, a tabular bit, and an increment. If the tabular bit specifies indirect addressing, then the special register so designated is used to address a main memory location in any bank indirectly. In this manner, any of the 256 special registers may be used to address any memory location indirectly. This type of address, called an indexed indirect memory location address, takes the form

Index Register Designator, N, Special Register Designator, Increment

As with any other indexed address, the index register is one of the controlling group and is designated by a number from 0 to 7. The special register designator may be absolute (from 0 to 31) or mnemonic, while the increment may be a number from 0 to 3 or may be omitted. An address of this type is interpreted by the machine in the same manner as an indexed special register address, except that the 12-bit configuration formed by ARGUS contains a tabular bit of 1 to indicate indirect addressing. The low-order eight bits of this configuration modify the low-order eight bits of the index register contents with carry, and the tabular bit in the result indicates whether the special register is addressed directly or used to address a memory location indirectly. As with indexed special register addresses, it is simplest to use an index register containing only a sign and a group indicator and otherwise all zeros. In this case, the special register and increment written by the programmer will be used and indirect addressing is assured. If other information is stored in the index register, the eight-bit addition process may alter the tabular bit. If this occurs, an IR, N address will produce the effect of an IR, Z address and vice versa.

Assume that the programmer writes the address

3, N, AU1

and that index register 3 contains only a sign and a group indicator. This group indicator and

the mnemonic designator AU1 define one of the 256 special registers, which is used to address a memory location indirectly. Since no increment was written, the contents of the special register are left unchanged.

Inactive Address

An inactive address is denoted in ARGUS language by a hyphen (-). This type of address may be used to gain access to three non-addressable registers called the accumulator, the mask register, and the low-order product register. In an addition instruction, for example, inactive addressing may be used to gain access to the accumulator. If the A and B address groups are inactive, the contents of the accumulator are transferred to C; if the B and C address groups are inactive, the contents of A are transferred to the accumulator. In similar fashion, inactive addressing may be used with the extract instruction to gain access to the mask register and with the "transfer A to B, go to C" (TS) instruction to gain access to the low-order product register. Inactive addressing is discussed more fully in the Programmers' Reference Manual.

Stopper Address

When a main memory address, stored in a special register, is modified by incrementing or augmenting, a carry may occur from the subaddress into the bank indicator. Thus a sequencing counter can be stepped through successive memory banks, and a single instruction can handle a record which is not stored entirely within one memory bank. There is one address, however, which by definition is neither incremented nor augmented when it appears in a special register. This address, called the stopper address, represents the highest-numbered location in the memory of a given Honeywell 800 system, regardless of the number of banks in the system (i.e., subaddress 2047 in the highest-numbered bank of the system). The stopper location can be utilized, for example, in a read instruction to move tape without disturbing the contents of memory or to read a portion of a tape record, discarding the balance. Due to relocation considerations, the stopper location can only be addressed through a special register in ARGUS language. This is accomplished by writing the symbolic tag STOPPER in a special address constant (see Section IX) and storing it in a special register. ARGUS replaces this tag with the address of the stopper location of the machine on which the program is to be run.

Numbers in Address Fields

ARGUS will convert any number up to 2047 appearing in an address field into binary. This ability should be used with caution, especially if the program is to be relocated for parallel processing.

SECTION VI

PROGRAM STRUCTURE

An assembled program may be divided into segments to conform to subdivisions of program logic. This makes it possible to have the coding for one function in memory while the coding for other functions remains on tape until it is needed. Segmentation makes efficient use of available memory storage and increases the number of programs that can be processed in parallel. Segments may be further broken down into subsegments to increase the flexibility of relocation by the Executive System, to provide communication among segments, and to exercise control over the allocation of memory as performed by the Assembly Program.

Segmentation

A segment is any part of a program which is loaded into memory and executed as a unit. Segmented programs fall into two general categories. One segment may operate upon the output of a previous segment with no internal communication, as in the case of a card-to-tape conversion which is followed by a sort and then by an updating run. Such segments resemble a series of separate programs run one after another. On the other hand, there may be communication among segments. The communication link may be either a control program which decides what segment to load next, or an area of memory containing data which varies from segment to segment, or both. Thus, unlike the first category, the order and frequency of executing the segments may not be predictable, but may depend upon the input data.

Programs in the first category are easily divided into mutually independent segments. If these segments have different equipment requirements, they can be scheduled for production more efficiently if they are written as separate programs. Programs in the second category, on the other hand, must be separated into interdependent and independent portions, according to the amount of memory available and the relative frequency of executing the various portions of coding. The programmer uses the control instructions PROGRAM and SEGMENT (see Section XI) to segment a program.

Any part of any segment may be specified as "common" (as described below under "Subsegmentation"). When part of a segment is common, the area of memory which it uses is reserved in all segments. It may be overlaid by common portions of other segments

only under the programmer's control. The portions of segments not specified as common, on the other hand, are overlaid by other segments under control of Executive. In other words, the only parts of the program guaranteed to be in memory during the execution of one segment are those words belonging specifically to that segment and any common portions of other segments which occupy the communication area at this time. For this reason, the symbolic tags defined in the common portions of a program may be referenced from any segment, while those not in a common area are available only to the segment in which they are defined.

Segment Loading

The name of the first segment which is to be executed is specified on the END card (see Section VIII). Executive automatically loads this segment when the program is initiated. The starting address of this segment must be loaded into the sequence counter by means of a SPEC constant with "Z, SC" in the location field.

The programmer uses a macro instruction called read segment to direct the automatic loading of all segments following the first. This instruction is logically equivalent to a transfer of control to that segment. The Executive System loads the requested segment and transfers control to a location specified in the macro instruction, under control of the sequence counter. Thus, no segment except the first one to be executed need load the sequence counter. If a segment does load the counter, the address specified in the read segment instruction will override the one loaded.

The format of the read segment instruction is:

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE ____ OF ____

	LOCATION 1–10	COMMAND CODE 11–22	S/C 23	A ADDRESS 24–37	B ADDRESS 38–51	C ADDRESS 52–65	REMARKS 66–80 (LINE NUMBER 73–74)
1	(TAG)	L, READSEG		NAME	START		

"Name" is the segment name specified in the PROGRAM or SEGMENT card, and "start" is the symbolic tag of the location in that segment where control is to be transferred. As in all macro instructions, the symbolic tag in the location field is optional.

The symbolic tag in the B address field must be a link tag, since it is referenced in one segment, while it actually belongs to another; this is an exception to the rule that all references within a segment must be to words within that segment or within the common area. Section IV states that a link tag is preceded by an "L" and a comma when it is defined. Such a tag may be referenced from any segment, by means of the read segment

macro instruction. It should be noted that a reference to this tag from another segment must not include address arithmetic. However, within the segment to which it belongs, this tag may be treated just as any other.

If desired, a segment may contain more than one starting location. Each starting location must be designated by means of a link tag. Different read segment macro instructions may be used to effect transfer of control to the different starting locations under various conditions.

Subsegmentation

A subsegment is a group of words within a segment which must retain the same relationship to each other in memory; the relationship of one subsegment to other subsegments within the segment is immaterial. Each segment may contain a maximum of seven subsegments. The division of a segment into subsegments is indicated by means of the control instruction SETLOC (see page 53). If no division is specified, the entire segment is considered to consist of one subsegment.

All subsegments of a segment occupy memory at the same time. However, during relocation each subsegment may be moved in memory independently of the other subsegments with the following exceptions:

1. If any subsegment crosses a bank boundary, all subsegments of every segment within the entire program will retain their original relationships to each other and to the bank boundaries. That is, the program is moved only by bank.
2. Two subsegments written to occupy the same memory bank will continue to share one memory bank unless one of the subsegments is in the communication area. In this case, the subsegments may be moved into two different banks.

The programmer may reference any word in a subsegment from any other portion of the same segment, according to the following rules:

1. Any reference to a word in another bank must be made via a special register.
2. Unless some subsegment crosses a bank boundary (so that the program is relocated by bank), all references to the communication area from outside this area must be made via special registers.
3. Address arithmetic must not be applied to a tagged word in one subsegment in order to reference a word in another subsegment unless the program is relocated by bank.

One of the primary reasons for designating a portion of coding or data as a subsegment is to specify that it should be made common to all segments. Subsegments not specified

as common are called normal subsegments. The normal subsegments of different segments are completely independent of each other; i.e., a subsegment numbered "1" in one segment bears no relation to a subsegment numbered "1" in another segment unless it is designated as common. If a subsegment is designated as common, however, its number will refer to the same subsegment in all segments.

If a subsegment has been designated as common in one segment, words can be added to it or overlaid on portions of it by other segments. The new words are preceded by a SETLOC instruction which states whether they are to be overlaid at a specified location or added at the end of the subsegment.

Another reason for dividing a segment into subsegments is to increase the flexibility of relocation. If two portions of a program (e.g., coding and data) need not occupy the same memory bank, it is desirable to code them as two subsegments, one in each of two banks. Executive may then relocate them into any space which is available.

Allocation

The Assembly Program assumes responsibility for the allocation of memory to a program; however, in some cases the programmer needs to have control over this allocation. For example, in a multi-bank program he may wish to place his masks in a particular bank so that he may refer to them directly from coding in that bank.

Each subsegment may contain any or all of these elements:

1. In-Line Coding - This consists of all program words except those which are designated as masks, loaded directly into special registers, or marked as out-of-sequence words. As each in-line word is processed by the Assembly Program, it is assigned to the next available location in the subsegment. Breaks in this sequence and/or initial values of this sequence may be specified by SETLOC instructions.
2. Out-of-Sequence Coding - This consists of all words in the subsegment which have "X, " in the location field (see page 20). These words are assigned locations starting immediately after the in-line words of that subsegment. Any subsegment which contains out-of-sequence coding should be stored entirely within one memory bank for ease of referencing such coding.
3. Masks - Masks are assigned in groups by means of the control instruction MASKGRP (see page 59). This instruction may designate a subsegment in which the groups named are to be placed; otherwise, they are placed in the subsegment in control at the end of the segment. If the SETLOC instruction for the subsegment containing the mask groups is immediately followed by a SETLOC for another subsegment, the mask groups will be

allocated in a subsegment by themselves.

4. Subroutines - If a subroutine is called for within a common subsegment, it is stored within that subsegment; otherwise, it is stored in the subsegment in control at the end of the segment. The programmer may use a SETLOC instruction immediately following the last line of a segment to specify the subsegment in which non-common subroutines are to be stored. All subroutines stored in a subsegment are allocated immediately following the last location used in the subsegment. The order in which subroutines are stored at the end of a subsegment is determined by the Assembly Program on the basis of their size, and not on the order in which they are called or the order in which they appear in the library.

If a segment consists of more than 2048 words, so that it must occupy more than one bank of memory, it can be subsegmented in such a way that the out-of-sequence words and masks which are referenced by one section of coding will be stored in a bank with that coding.

Relocation

Because a program is prepared without specific knowledge of the actual memory and equipment which will be assigned to it, certain precautions must be observed during program preparation to facilitate successful relocation and operation of the program. Since the program will undoubtedly be run in parallel with others at some time, observance of the facts of relocation outlined below is also necessary in order that the program may not interfere with others.

1. The relocatable quantities - group and bank indicators and peripheral codes - should not be treated as numerical values. In other words, no arithmetic operations should be performed on these quantities.

2. Although a one-to-one correspondence exists between group indicators of a program before and after relocation, this is not so with bank indicators. Different bank indicators may be assigned the same value during relocation, since subsegments written for different banks may be loaded into the same bank at run time. However, subsegments written for the same bank will always be assigned to the same bank, with the exception of common subsegments.

3. There is no relationship between bank indicators of different segments, except for common subsegments which remain in the same absolute areas throughout execution of the program. Therefore, references between common and normal subsegments must be made via special registers.

4. Statements (2) and (3) above are limited to programs relocatable modulo 64, not to those relocatable by bank only.

5. Bank, group, and control unit indicators can be identified properly only if constants to be loaded into special registers are special address (SPEC) or complete address (CAC) constants (see Section IX).

6. Fixed, non-relocatable locations (e.g., date location and inquiry stations) must be addressed via special registers. The stopper location, which is represented by the tag STOPPER in a SPEC constant (see page 28), must also be addressed via a special register.

7. Because group indicators and tape control unit identifications are relocatable, care must be used in writing a program control constant to examine the program control register. A program should examine only group and buffer bits used by that program.

8. The MPC instruction (see page 50) must be used with great care. In particular, only those groups used by the program should be altered in any way.

9. The special registers RAC, DRAC, WAC, and DWAC are associated with control unit indicators and not with group indicators. They must be addressed via special registers. When they are addressed, control unit assignments must be uniquely defined. It must be remembered that these counters may contain information from other programs using the same control unit.

In those control groups which contain the read-write counters RAC, DRAC, WAC, and DWAC, special registers S4 through S7 are not available. Relocation is facilitated by always specifying a group in which these registers are unavailable, unless they are actually required by the program. However, if a program does use any of the special registers S4 through S7, it is the programmer's responsibility to specify a group in which these registers are available. Further information on relocation can be found in the Executive System Manual.

EXAMPLE: Two segments of a program have been assembled and allocated in memory as shown in the left-hand two columns of Figure 7 (Before Relocation). Segment A consists of a common subsegment (numbered 1) and four other subsegments. Segment B consists of a common subsegment (which is also numbered 1) and three other subsegments. This program has been executed and checked out, using memory banks 0 through 3, as shown. Note that part of the common subsegment is loaded with segment A and part with segment B and that these parts are assigned overlapping memory areas.

The same program is to be loaded for processing in parallel with a number of other programs which are using memory banks 0, 1, and 4 through 7. When this program is scheduled for a production run, Executive examines its relocation information and relocates it as shown in the right-hand two columns of Figure 7 (After Relocation), so that it can be processed entirely within the available memory banks 2 and 3. A comparison of the two halves of Figure 7 reveals that the following rules govern the relocation process.

1. All subsegments are relocated in integral multiples of 64 locations to preserve all mask group relationships;

2. The two portions of common subsegment 1 retain the original overlapped relationship; and

3. Normal subsegments 2 and 4 of segment A, originally sharing the same memory bank, continue to share a bank after relocation. These subsegments may communicate by means of direct addressing. All other communication between subsegments must be by means of indirect addressing.

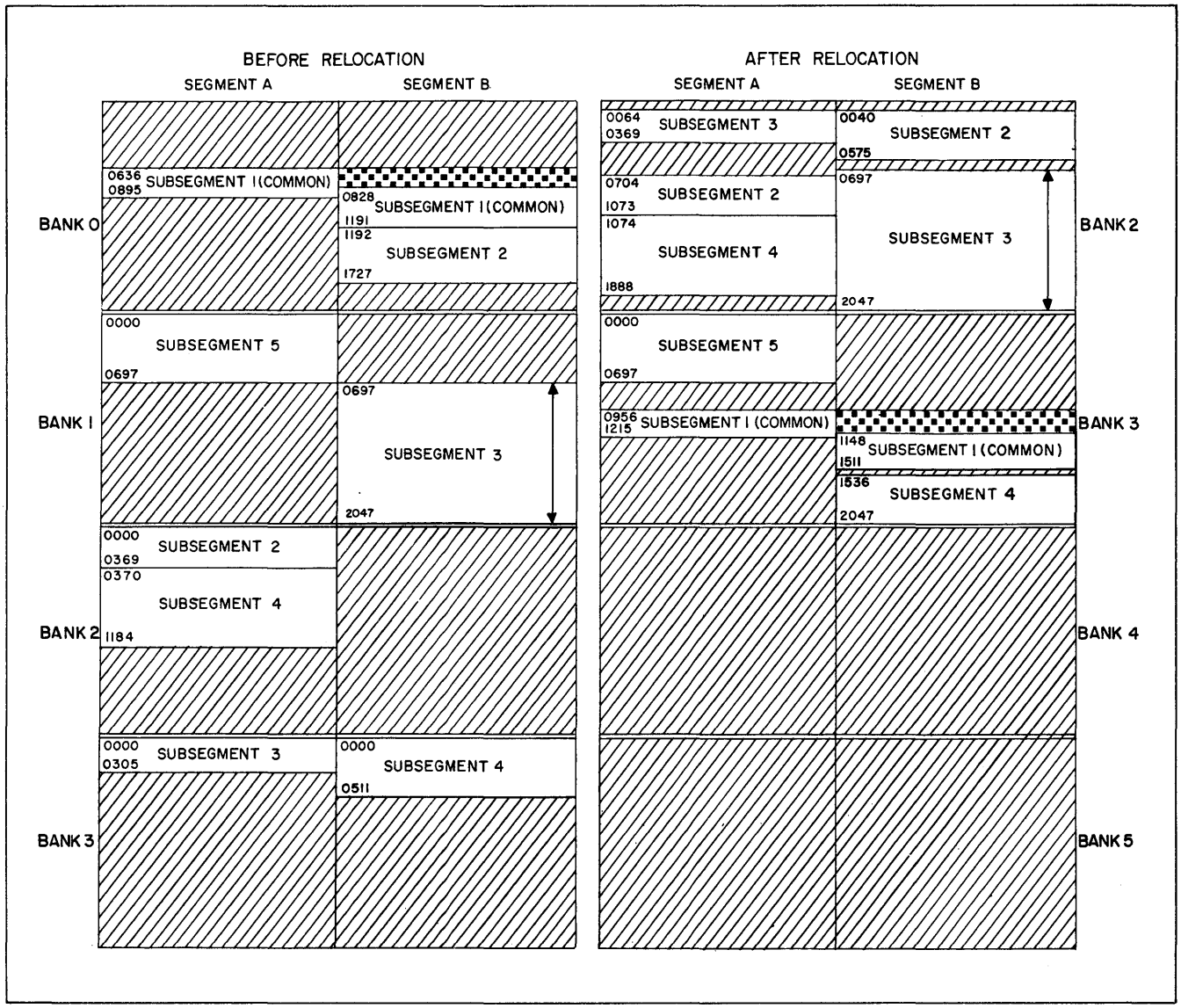


Figure 7. Example of Program Relocation

SECTION VII

MACHINE INSTRUCTIONS

Honeywell 800 machine instructions are specified in ARGUS language using the mnemonic operation codes shown in Figure 8. Note that ARGUS recognizes the five classes of machine instructions, namely, general, peripheral, shift, scientific, and simulator instructions, plus a group of extended instructions. The functions of Honeywell 800 machine instructions are summarized in Appendix E. The reader will find it convenient to refer to Appendix E for the details of the machine instructions illustrated on the following pages.

A machine instruction consists of a command code group and three address groups. The command code group may include such information as a mask tag or a peripheral code in addition to the mnemonic operation code, depending upon the instruction type. An address group may refer to a memory location or a special register, using one of the address forms given in Figure 6, or it may contain a parameter dictated by the format of the instruction. Information written either in the command code field, or in any address field may be punched anywhere in the indicated card field; spaces in these fields are ignored when assembling machine instructions.

General Instructions

The instructions in this class perform such operations as arithmetic, information transfers, comparisons, program control, and information checking. All of these instructions, with the exception of proceed, have the ability to designate the source of the following instruction. If column 23 contains an "S" or is blank, the address of the next instruction is obtained from the sequence counter; if this column contains a "C", the address of the next instruction is obtained from the cosequence counter. Column 23 of a proceed instruction is not used, and the following instruction is always selected by the sequencing counter which selected the proceed instruction.

Three examples of general instructions are shown on the following page. The first instruction adds the contents of location PRICE to the contents of the location three after PRICE and stores the result in location AMTDUE. Both operands and the result are regarded as signed 11-digit decimal numbers. The second instruction transfers 10 words from consecutive memory locations starting at INPUT to consecutive locations starting at WORK1. The third compares numerically the contents of a memory location reached by indexed addressing

with the contents of COUNTER. If (3, 7) is less than or equal to (COUNTER), the cosequence counter is reset to the memory location 14 beyond the location of this instruction. (Parentheses are used around an address to specify the contents of the indicated location.) Each of the first two instructions is followed by an instruction selected by the sequence counter; the third instruction designates the cosequence counter as the source of the next instruction, whether or not the comparison is satisfied. The functions of these three instructions can be verified by referring to Appendix E.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION 1–10	COMMAND CODE 11–22	S/C 23	A ADDRESS 24–37	B ADDRESS 38–51	C ADDRESS 52–65	REMARKS 66–80 (LINE NUMBER 73–74)
1		DA		PRICE	PRICE+3	AMTDUE	
2		TN	S	INPUT	10	WORK1	
3		LN	C	3,7	COUNTER	C,+14	

SEQUENCE CHANGE INSTRUCTIONS: Several instructions have the ability to execute a programmed change of sequence by placing the C address in the sequencing counter specified as the source of the next instruction. An example is the instruction TS (transfer A to B and go to C). In such instructions, the C address may take any valid address format. However, if a special register is addressed, the value of the tabular bit is ignored and the result is always a memory location address. Thus, a direct or indexed special register address, if used as a change of sequence, will be interpreted respectively as an indirect or an indexed indirect memory location address.

FIELD INSTRUCTIONS: Many of the instructions in the general class can be performed under the control of masks, which allow them to designate partial words as operands and as results. These instructions, which are indicated by a superscript^2 in Figure 8 and in Appendix E, are called field instructions. When a field instruction is masked, the same mask is applied to operands and results. Only those bit positions in the operands which correspond to binary ones in the mask, called the masked portions, are used. All field instructions are masked protectively; i.e., the unmasked portions of the result locations are not altered by the operation.

The mask to be used in a field instruction may be designated by writing its symbolic tag in the command code field, following the operation code and separated from it by a comma. A mask whose mask indicator is F (for field instructions) or B (for both field and shift instructions) may be designated in a field instruction. If the tag which follows the operation

General	
DA[2]	Decimal Add
DS[2]	Decimal Subtract
DM	Decimal Multiply
BA[2]	Binary Add
BS[2]	Binary Subtract
BM	Binary Multiply
WA[2]	Word Add
WD[2]	Word Difference
HA[2]	Half Add
BT	Binary Accumulate
DT	Decimal Accumulate
SM[2]	Superimpose
SS	Substitute
EX	Extract
TX[2]	Transfer A to C
TS[2]	Transfer A to B and go to C
TN	N-word Transfer
MT	Multiple Transfer
IT	Item Transfer
RT	Record Transfer
NN[2]	Inequality Comparison, Numeric
NA[2]	Inequality Comparison, Alphabetic
LN[2]	Less Than or Equal Comparison, Numeric
LA[2]	Less Than or Equal Comparison, Alphabetic
PR	Proceed (no operation)
CC	Compute Orthocount
CP[2]	Check Parity
MPC[3]	Control Program
Shift	
SWE	Shift Word, Extracting
SWS	Shift Word, Substituting
SPE	Shift Preserving Sign, Extracting
SPS	Shift Preserving Sign, Substituting
SSL	Shift and Select
Simulator	
S	Simulator

Peripheral	
RF	Read Forward
RB	Read Backward
WF	Write Forward
RW	Rewind
Extended	
PRA	Print Alphanumeric
PRD	Print Hexadecimal
PRO	Print Octal
STOP	Stop This Program
DOFF	Stop Specified Programs
DON	Start Specified Programs
SCON	Give Control of Specified Programs to Sequence Counters
CSCON	Give Control of Specified Programs to Cosequence Counters
SPCR	Transfer (PCR) to C
Scientific[1]	
FBA	Floating Binary Add
FBS	Floating Binary Subtract
FBM	Floating Binary Multiply
FBD	Floating Binary Divide
FDA	Floating Decimal Add
FDS	Floating Decimal Subtract
FDM	Floating Decimal Multiply
FDD	Floating Decimal Divide
FLN	Normalized Less Than Comparison
FNN	Normalized Inequality Comparison
FFN	Fixed to Floating Normalize
FBAU	Floating Binary Add, Unnormalized
FDAU	Floating Decimal Add, Unnormalized
FBSU	Floating Binary Subtract, Unnormalized
FDSU	Floating Decimal Subtract, Unnormalized
FBAE	Floating Binary Addition, Extended Precision
FBSE	Floating Binary Subtraction, Extended Precision
ULD	Multiple Unload
BD	Fixed Binary Divide
DD	Fixed Decimal Divide

1. These instructions are included in Floating-Point Option; machines not equipped with this option may implement them as library pseudo instructions.
2. These instructions may reference type F or type B masks or may direct mask generation.
3. Most MPC functions represented by extended instructions STOP, DOFF, DON, SCON, CSCON, and SPCR.

Figure 8. ARGUS Mnemonic Operation Codes for Honeywell 800 Machine Instructions

code has both a mask assignment and a complex assignment, the mask assignment is used. The method of assigning masks in groups of consecutive memory locations is described in Section X.

Alternatively, the programmer may direct ARGUS to generate the desired mask. The following three items of information in the command code field, separated from the operation code and from each other by commas, direct the generation of the desired mask by ARGUS:

(M_1) The position of the high-order character in the masked field. This may be a number from 1 to 8 for alphanumeric characters, from 1 to 12 for unsigned hexadecimal digits, or from 2 to 12 for signed hexadecimal digits (see Figure 2).

(M_2) The number of characters in the masked field. This may be a number from 1 to 8 for alphanumeric characters, from 0 to 11 for signed hexadecimal digits, or from 1 to 12 for unsigned hexadecimal digits.

(M_3) A character to specify the bit position(s) containing the sign of the masked field. This character may be a number from 1 to 4, corresponding to the four sign bits from left to right, or it may be an "S" to specify the use of all four as the sign of the masked field. If the masked field is unsigned, as in alphanumeric information, this character is a 0 or is omitted.

The use of generated masks is limited to alphanumeric and hexadecimal fields of consecutive characters. Tags must be used to designate masks for binary fields or for fields of non-consecutive characters. The type of field, alphanumeric or hexadecimal, is implied by the operation code in most cases. Arithmetic operations always involve numeric words and the comparison instructions specify numeric or alphabetic comparison.[1] In certain instructions, however, the type of field is ambiguous. If one of these instructions (viz., WA, WD, HA, TS, TX, SM and CP) is to be performed with a generated mask, a three-character operation code must be formed by appending an "A" for alphanumeric or a "D" for hexadecimal to the two-character code shown in Figure 8. Both designated and generated masks are illustrated in the following examples.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION (1–10)	COMMAND CODE (11–22)	S/C (24)	A ADDRESS (24–37)	B ADDRESS (38–51)	C ADDRESS (52–65)	REMARKS (66–80)
1		DS, PAYROLL 3	C	GROSS PAY	GROSSPAY - 3	NETPAY	
2							
3		TXA, 1, 7		NAME		PRINTOUT	
4							
5		LN, 2, 8, S	S	TOTAL	DATA 1	COMPUTE	

[1] In generating a mask for an alphabetic comparison, ARGUS assumes alphabetic operands. If operands of such an instruction are numeric, any mask used must be designated by a symbolic tag.

The first instruction above subtracts decimally the contents of the location three before location GROSSPAY from the contents of location GROSSPAY and stores the result in location NETPAY. Assume that the mask designated as PAYROLL3 has the configuration

G00 000 GGG GGG

in hexadecimal form. This mask is applied to the subtraction operation, with the result that only the sign and the low-order six digits of each operand are considered and only those digit positions are affected in location NETPAY.

The second instruction transfers (NAME) to location PRINTOUT. The contents of the command code field direct ARGUS to generate an alphanumeric mask of seven characters, starting with character one (the left-most character). Thus, only the first seven characters of NAME are transferred and the eighth character position in location PRINTOUT is not altered.

The third instruction above compares (TOTAL) with (DATA1) numerically. ARGUS generates a hexadecimal mask (because a numeric comparison is specified) which masks eight digits starting with digit 2, the digit immediately following the sign. All four bits of digit 1 are designated as sign bits. If the masked portion of TOTAL is less than or equal to the masked portion of DATA1, the sequence counter (specified in the S/C column) is reset to COMPUTE.

Field instructions are subject to the restriction that when they are masked, they can neither address special registers nor use them to address main memory indirectly. Consequently, they must obtain their operands and store their results by means of either direct or indexed addressing of memory locations. This restriction does not apply to the remainder of the general instructions or to field instructions performed without masks.

N-WORD INSTRUCTIONS: Four general instructions which use the B address field to specify a number of words to be transferred (from 0-63) are the binary and decimal accumulate, n-word transfer, and multiple transfer instructions. In any of these instructions, the B address field may contain a symbolic tag (with or without address modifier) which is equated elsewhere to a number, by means of an EQUALS instruction (see Section VIII). The value of the tag (or the value of the modified tag) must be in the range 0 through 63. For example, if a block of data 20 words long is to be manipulated by several different n-word instructions, the tag BLOCK might be equated to the value 20. Then the following instruction could be used to transfer the data from locations starting with INPUT to locations starting with OUTPUT.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	66 REMARKS LINE NUMBER 73	74 80
	TN		INPUT	BLOCK	OUTPUT		

It is only necessary to modify the instruction which defines the tag BLOCK, rather than modifying all of the n-word instructions involved, if the length of the data block changes.

Peripheral Instructions

Every instruction in this class performs some operation involving a magnetic tape unit or a terminal device. Peripheral instructions are subject to the same addressing restrictions as masked field instructions. They cannot specify a special register address or an indirect memory location address in any address field. Furthermore, instructions in this class lack the provision for specifying the source of the following instruction. Therefore, the S/C subfield (column 23) is not used in a peripheral instruction, and the address of the following instruction is always taken from the same sequencing counter that selected the peripheral instruction.

The command code field in a peripheral instruction contains a two-character operation code followed by a comma and an alphabetic peripheral code from AA to HH. The assignment of peripheral codes to magnetic tape units and terminal devices is established individually at each Honeywell 800 installation. In the case of a terminal device, the second letter of the peripheral code designates the device type, according to the following convention:

A = card reader

B = printer

C = card punch

D = paper tape reader

E = paper tape punch

In the case of a magnetic tape unit, the second character may be any letter from A to H. The Assembly Program uses this convention to analyze the peripheral requirements of a program and to diagnose and report any attempt to address a peripheral device which is not capable of performing the requested operation (e.g., a rewind addressed to a card reader). Every Honeywell 800 installation is provided with a table of peripheral code assignments.

The A address field in a peripheral read or write instruction specifies the location into which the first word is to be read or from which the first word is to be written. The read address counter (RAC) or the write address counter (WAC) directs the reading of subsequent

words into or the writing of subsequent words from consecutive higher-numbered locations until an end-of-record word is encountered (see Appendix E). (In a read backward instruction, the RAC directs the reading of subsequent words into consecutive lower-numbered locations.)

If the B address field in a read or write instruction to magnetic tape is active, the operation is a distributed read or write, and the record read or written is sensed for end-of-item symbols (see Appendix E). In this case, the B address field specifies the starting location of a stored table, which in turn contains the starting addresses of memory areas into which the items of a record are to be distributed or from which items are to be assembled to form a record. The first item is read or written, starting at A; subsequent items are read or written starting at the addresses stored in the table. The distributed read address counter (DRAC) or the distributed write address counter (DWAC) directs the selection of addresses from the stored table to distribute or assemble the items of a record. (If a read backward is distributed, the B address field specifies the final location of a stored table of final addresses of items.) If the B address field is inactive, the operation is a normal read or write, end-of-item symbols are not sensed, and the DRAC and DWAC are not used.

The C address field in any read or write instruction may be used to specify a change in the contents of the sequencing counter which selected the instruction; if the C address field is inactive, no change of sequence takes place. If the C address is active, it is interpreted as in any other sequence change instruction (see above).

If the A address field in a rewind instruction is active, the rewound tape is interlocked against further peripheral operations. The B and C address fields in a rewind instruction are not used.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	66 REMARKS (LINE NUMBER) 73	74 80
1		WF, FB		UPDATE	—	READIN+2		
2								
3		RF, AB		TRANSACT	WORK 3	—		

The function of the first instruction above is to write one record or print one line on device FB, depending upon whether this device is a magnetic tape unit or a printer. The record to be written is stored in memory starting at location UPDATE. Since the B address field is inactive, end-of-item symbols are not sensed; i.e., the record is assumed to be stored in consecutive memory locations. The C address field designates that the counter which selected this instruction is to be set to address READIN +2.

The second sample instruction reads one record from device AB. The record is to be stored in memory, the first item starting at location TRANSACT. WORK3 is the first location of a stored table of starting addresses of items. As the record is read, end-of-item symbols are sensed, and the RAC and DRAC control distribution of the remaining items to non-consecutive memory areas. As terminal devices cannot perform distributed reading, AB must be a magnetic tape unit. Since the C address field is inactive, the sequencing counter which selected this instruction is incremented normally to form the address of the next instruction.

Shift Instructions

Four of the five shift instructions are used to alter the positions of data fields within words. Two of these substitute the shifted field into a word which is otherwise unaltered; the other two extract the shifted field into a word which is otherwise cleared to all zeros. The fifth instruction, shift and select, is used to select one of a possible 2048 locations as the source of the following instruction, based upon the value of a data field.

Every shift instruction is performed under the control of a mask. The location of the word to be shifted is written in the A address field. The type, extent, and direction of the shift are specified in the B address field. All five instructions perform end-around shifting; i.e., every character shifted out of a word at one end reappears at the opposite end. The shifted word is masked and then delivered to the location specified by C (or used to modify the C address in the shift and select instruction). The two shift and substitute instructions protect the unmasked portions of the result location. The two shift and extract instructions clear the unmasked portions of the result location to all binary zeros. As in the case of field instructions, the desired mask may be either designated symbolically or generated by ARGUS. The tag of a designated mask is written in the command code field of the shift instruction, following the operation code and separated from it by a comma. A mask with a mask indicator of S (for shift instruction) or B (for both) may be designated in a shift instruction. If the tag written has both a mask assignment and a complex assignment, the mask assignment is used. To generate a mask, ARGUS uses the same three items of information (M_1, M_2, and M_3) as outlined under field instructions. M_1, M_2, and M_3 follow the operation code and are separated by commas. Since shifting takes place before masking, M_1 must specify the position of the high-order character in the masked field after shifting. The shift word and the shift and select instructions do not normally include a value of M_3.

Any valid address format, as shown in Figure 6, may be used in the A or C address field of a shift instruction. However, the C address field of a shift and select instruction is

interpreted as in any other sequence change instruction (see page 38). The B address field of a shift instruction normally contains three items of information, separated by commas, which specify the nature and extent of the shift:

(B_1) A character to designate the type of characters to be shifted.

A	= six-bit alphanumeric
D	= four-bit decimal
B or blank	= binary

*(B_2) The number of positions that the word is to be shifted, from 0 to 8 for alphanumeric characters, from 0 to 12 for decimal digits, or from 0 to 48 for bits.

(B_3) A character to designate the direction of shift.

L	= left
R or blank	= right

Alternatively, the B address field may contain a symbolic tag (with or without address modifier) which is equated elsewhere to a number, by means of an EQUALS instruction (see Section VIII). The value of the tag (or of the modified tag) must be in the range 0 through 48. ARGUS interprets such a tag as the number of bit positions to be shifted to the right.

As in the case of field instructions, the use of generated masks is limited to alphanumeric and decimal fields of consecutive characters. Masks for binary fields or for fields of non-consecutive characters must be designated symbolically. If no mask information is written in the command code field and the shifted field is alphanumeric or decimal (B_1 = A or D), ARGUS generates a mask to suppress that portion of the word moved either right or left end around during the shifting process. However, if B_1 specifies a binary shift or the B address field is symbolic, ARGUS generates a mask of all ones in the absence of a mask tag in the command code field. The shift and select instruction requires a mask which allows no more than 11 low-order bits to be used in modifying the C address.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION (1–10)	COMMAND CODE (11–22)	S/C (23)	A ADDRESS (24–37)	B ADDRESS (38–51)	C ADDRESS (52–65)	REMARKS (66–80)
1		SPS, PARTNUMB	S	3, 9	B, 10, R	PART LIST +5	
2							
3		SWE, 6, 3		EMPLOYEE +4	D, 3	N, R3, 1	
4							
5		SSL, 11, 2	C	SELECTOR	D, 4	C, -5	

The first sample instruction above shifts the contents of the location specified by indexed address 3, 9 ten binary places to the right, preserving the sign, and stores the result in location PARTLIST +5, under control of a mask tagged PARTNUMB. The unmasked portion of the

*If B_1 and B_3 are both blank, B_2 may be as large as 63.

result location is protected. The sequence counter is consulted for the source of the next instruction.

The second instruction shifts the contents of location EMPLOYEE +4, including the sign, three decimal places to the right (B_3 is blank) and stores the result in the location specified by indirect address N, R3, 1. The generated mask produces an unsigned field of three decimal digits beginning with digit 6 and replaces the remainder of the result location with all 0 bits. Again the sequence counter is consulted for the source of the next instruction.

The third instruction shifts the contents of location SELECTOR, including the sign, four decimal places to the right under control of a generated mask which produces a field of two low-order decimal digits. These eight bits are added in binary form to the address of a memory location five before the location of this instruction (since this is not marked as an out-of-sequence word). The modified address is then stored in the cosequence counter which is designated as the source of the next instruction.

Scientific Instructions

This class includes the instructions which perform arithmetic operations and comparisons on floating-point numbers. Figure 2, page 8, shows that a Honeywell 800 floating-point word consists of a 40-bit mantissa, a seven-bit exponent, and a sign bit. This configuration may represent either a decimal number or a binary number in floating-point form. Arithmetic instructions are provided to handle floating-point words either as decimal or as binary numbers. Data which is to be manipulated in floating-point form is normally assembled in this form, using the floating-point binary and floating-point decimal constants described in Section IX. However, fixed-point binary and decimal constants can be converted to floating-point form. In normalized floating-point decimal form, the exponent represents a power of 10 from the -64^{th} to the $+63^{rd}$ and the mantissa a 10-digit number from .1000 to .9999----. In normalized floating-point binary form, the exponent represents a power of 16 from the -64^{th} to the $+63^{rd}$ and the mantissa a 40-bit number from .00010000---- to .11111111----. An exception is the value 0. Although any floating-point number whose mantissa is 0 has the value of 0, a normalized floating-point 0 in the Honeywell 800 is defined as a number having a positive sign and all binary zeros in the exponent and the mantissa.

The operands used in a floating-point instruction must be in floating-point form but not necessarily normalized (with the exception of divisors and operands for the comparison instructions). The results are in correct floating-point form, and are normalized except where

otherwise specified. Exponential overflow occurs if the exponent of the result exceeds +63; exponential underflow occurs if the result exponent is less than -64. When exponential overflow is sensed, an unprogrammed transfer of control to U + 14 or U + 15 is executed, where U represents the location whose address is stored in the unprogrammed transfer register (see Figure 5, page 19). When exponential underflow is sensed, the unprogrammed transfer is to U + 12 or U + 13.

A floating-point divide instruction cannot be executed if the possibility exists that the divisor is 0. A fixed-point divide instruction cannot be executed if the absolute value of the quotient equals or exceeds unity. In either case, an unprogrammed transfer of control to U + 10 or U + 11 is executed.

The machine logic to implement the scientific instructions is an optional feature of the Honeywell 800. Included in this option are the two fixed-point division instructions. Though none of these can be performed as machine instructions on systems which do not include the floating-point option, they are all represented by library routines which can be performed by such systems. One of the items of input required by ARGUS is an indication of whether or not programs are to be assembled for a system which includes the floating-point option (see Section XI). In assembling programs for such a system, scientific instructions are assembled as machine instructions; otherwise, they are handled as library routine pseudo instructions (as described in Section XIII).

Simulator Instructions

The Honeywell 800 complement of machine instructions is designed to perform the logical operations normally required for business data processing and scientific computation. In addition, the provision of simulator instructions permits the programmer to represent with a single instruction any function not built into the equipment logic, such as a machine instruction for some other data processing system.

For each simulator instruction, the programmer codes a simulator routine which is stored elsewhere in memory. The control instruction SIMULATE (see Section VIII) must precede the simulator routine and must be tagged in its location field. The simulator instruction sets up a transfer of control to this routine as well as a means of returning control to the main program.

The command code field of a simulator instruction contains an "S" followed by a comma and the address of the SIMULATE instruction which precedes the desired routine. The S/C

column is not used. The address fields may contain parameters required by the routine. In particular, the contents of the A and C address fields are stored as complete addresses in special registers AU1 and AU2. If either or both of these parameters is to be indirectly addressed via the appropriate special register, it must be either a direct or an indexed memory location address. Otherwise, each address field may contain any parameter which can be expressed as a decimal number less than 2048. Decimal parameters are converted to binary by ARGUS.

The desired routine may be specified by either direct or indexed addressing. Direct addressing can only be used to execute a routine stored in the same memory bank as the simulator instruction. In this case, the programmer writes the tag of the SIMULATE instruction in the command code field of the simulator instruction. Indexed addressing must be used in the more general case to execute a simulator routine from any bank of memory. The index register to be used is loaded with the tag of the SIMULATE instruction and an address modifier of -7, using the special address constant (SPEC) described in Section IX. The same index register is then referenced with an augmenter of 7 in the command code field of the simulator instruction. (The Honeywell 800 recognizes a simulator instruction by the presence of three low-order binary ones in the command code; hence the necessity of modifying the tag of the SIMULATE instruction by -7 and then augmenting the result by +7 in the simulator instruction command code.)

When a simulator instruction is executed, the instruction itself is transferred to the location specified in its own command code field. This is the location which immediately precedes the desired routine. ARGUS assures that it is a location whose subaddress contains three low-order binary ones, as required by the above definition of a simulator instruction. The cosequence counter is loaded with the starting address of the routine, and the contents of the source counter, after normal incrementing, are stored in the cosequence history register to provide a return to the main program.

For example, the control instruction

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

1 LOCATION 10	11 COMMAND CODE 22	S/C 24	A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	66 REMARKS LINE NUMBER 73	74 80
CUBEROOT	SIMULATE						

is followed by a simulator routine which performs a cube root computation. The tag CUBEROOT is assigned to the location immediately preceding the start of the routine. The

operand location and the result location of the cube root computation, which are written in the A and C address fields of the simulator instruction, may be indirectly addressed by referencing AU1 and AU2, respectively. The cube root routine may be executed from the memory bank in which it is stored by writing an instruction in the program of the first sample form shown below. To execute this routine from any memory bank, the programmer must load a special address constant of CUBEROOT -7 into an index register and write an instruction of the second sample form shown below.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C 24	A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	REMARKS 66 LINE NUMBER 73	74 80
1		5 CUBEROOT		7, 15		COMPUTE +11		
2								
3		5, 3, 7		7, 15		COMPUTE+11		

When either of these instructions is executed, it is transferred to location CUBEROOT, the cosequence counter is set to CUBEROOT +1, and the contents of the source counter are stored in the cosequence history register for use as an exit to the main program. The indexed address of the operand is 7, 15 and the cube root is stored in location COMPUTE +11.

Multiprogram Control

The automatic parallel processing of up to eight programs is directed by a central processor element called multiprogram control which examines the group of eight program demand bits in a non-addressable register called the program control register. These bits represent the eight special register groups and specify the active or inactive status of each group. Normally, when a machine instruction is completed, these bits are examined and an instruction is initiated under control of the next active special register group in sequence. In the following discussion, this process is called hunting for another program demand.

All machine instructions cause multiprogram control to hunt for another demand with the following exceptions:

1. Any instruction, including a simulator instruction, which results in a programmed change of sequence;
2. Any instruction, such as multiply, which generates a two-word result;
3. An instruction which contains an inactive C address or an inactive result address, except rewind, which always causes multiprogram control to hunt for another demand;
4. An instruction which results in an unprogrammed transfer;
5. All program control instructions (see below) direct multiprogram control whether or not to hunt for another demand, except STOP which always causes hunting.

An instruction which inhibits hunting for another demand is always followed by another instruction from the same program. This feature is normally used to store the contents of a non-addressable register which might be destroyed by another program.

Extended Instructions

There are two cases in which a group of ARGUS machine instructions is represented by a single machine language operation code. These so-called extended instructions are the print and the program control instructions. Each ARGUS extended instruction has its own mnemonic operation code. The corresponding function is uniquely designated in machine language by an operation code plus a specified portion of an address field. Thus, an ARGUS extended instruction represents the corresponding machine operation code plus the additional information required to designate the desired operation.

PROGRAM CONTROL INSTRUCTIONS: One of the non-addressable registers in the Honeywell 800 is called the program control register. Its contents represent the status of input and output buffer interlocks, the demand conditions of the various special register groups, and the sequencing counter designated to select the next instruction in each special register group. Access to the program control register is normally limited to the Executive System, in order to insure fully automatic parallel processing. However, the programmer can gain access to it by means of a machine instruction called control program. The B address of this instruction specifies one of eight different operations to be performed on the contents of the program control register, as well as the portion of these contents to be altered. The machine format of the control program instruction is described in the Honeywell 800 Programmers' Reference Manual.

Six of the eight operations which can be performed by the control program instruction are represented in ARGUS notation by a group of extended instructions called program control instructions. These six instructions perform all program control operations normally required by the programmer. In addition, in order to make all eight control functions available, ARGUS can accept the machine instruction MPC. The B address field of this instruction contains three hexadecimal digits which specify the desired control operation and the programs to be affected, as described in the Reference Manual.

The present discussion deals with the ARGUS extended instructions. Any of these extended operation codes may be followed by a comma and an "H" in the command code field if the system is to hunt for a demand from another program. Otherwise, with the exception of STOP, the current instruction is followed by another instruction from the same program.

After a STOP instruction, the system always hunts for another demand. The S/C subfield is used in the normal manner. The contents of the A address field, which may be any valid address format, are not used in executing a program control instruction. The B address field contains the numbers of up to seven programs, separated by commas, to be controlled by the instruction. An exception is the SPCR instruction which performs no control function and in which the B address field is left blank. The number of a program is the group indicator of the special register group controlling that program. Before a program control instruction is executed, the contents of the program control register are transferred to the location specified in the C address field. This field may contain any valid memory location address form, but it is interpreted as in a sequence change instruction (see page 38). If it is inactive, the contents of the program control register are not retained.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION 1–10	COMMAND CODE 11–22	S/C 23	A ADDRESS 24–37	B ADDRESS 38–51	C ADDRESS 52–65	REMARKS 66–80 (LINE NUMBER 73–74)
1		DOFF	S		2,3,4,6	C+3	
2							
3		SCON,H			1,4	N,R3,3	
4							
5		SPCR,H	C			CONTROL	

The first of these sample instructions transfers the contents of the program control register to the memory location three after the location of the instruction. Then the programs using special register groups 2, 3, 4, and 6 are turned off. The system is not directed to hunt for another demand but to execute another instruction in the same program, under control of the sequence counter. The second instruction stores (PCR) in an indirectly addressed location and turns over control of programs 1 and 4 to their respective sequence counters. The sequence counter is specified as the source of the next instruction in the same program and the system is directed to hunt for another program demand. The third instruction stores (PCR) in location CONTROL, transfers control of its own program to the cosequence counter, and directs the system to hunt for another program demand.

PRINT INSTRUCTIONS: From 1 to 47 automatic typewriters can be included in a Honeywell 800 system. The standard unit is located at the console and is referred to as the console typewriter. A second optional unit, known as the slave, is normally located somewhere near the control area. The provision of a slave typewriter allows program printouts to be physically separated from console input information. In addition, two programs operating in parallel can produce printout information on separate typewriters. Up to 45 optional remote typewriters can also be included in the system.

The machine instruction print is represented in ARGUS notation by three extended instructions: print alphanumeric, print hexadecimal, and print octal. Any of these operation codes may be followed in the command code field by a comma and an "M" (denoting more information to follow before carriage return) or an "MR" (denoting more information to follow after carriage return). If either of these carriage controls appears, the typewriter is interlocked against all other programs until another word is printed from the same program. If neither appears, the carriage is returned after printing and the typewriter is released to print from any program.

The A address field specifies the location of the word to be printed and may contain any valid address format. The B address field contains a "C", an "S", or a two-digit number specifying the typewriter which is to print. Either "C" or 00 indicates the console typewriter; "S" or 01 indicates the slave. A remote station may be specified by a number from 02 to 46, depending upon the number of such stations in the system. If the B address field is left blank, the console typewriter will print. The C address field may contain a programmed sequence change or it may be inactive. The contents of this field (if active) are interpreted as in any sequence change instruction (see page 38) and stored in the counter specified by the S/C subfield.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C 24	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	66 REMARKS LINE NUMBER 73	74 80
1		PRA, MR	S	C, +5	C	—		
2								
3		PRD	C	RESULT	05	1, 14		

The first sample instruction causes the console typewriter to print in alphanumeric form the contents of the location five after that of the instruction itself. The typewriter is interlocked to receive another print instruction from this program (after carriage return) and the next instruction is selected by the sequence counter. Since the C address is inactive, there is no programmed sequence change. The second instruction causes remote typewriter 05 to print in hexadecimal form the contents of location RESULT. The carriage is returned and the interlock released. The cosequence counter is changed to the contents of X1 augmented by 14, and control is transferred to this location.

SECTION VIII

ASSEMBLY CONTROL INSTRUCTIONS

The ARGUS assembly 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, although they are not assembled and do not result in the inclusion of any machine words in the program. Each of these instructions may be used as many times as required within a program.

SETLOC

The primary function of the SETLOC instruction is to direct the subsegmentation of a program segment. This function can only be accomplished by the use of SETLOC. The programmer may also use the SETLOC instruction to direct the allocation process by specifying a memory location address, a bank indicator, a group indicator, or any combination of these elements. To the extent that the programmer does not control allocation, this process is automatically handled by the assembly program.

The first SETLOC instruction which specifies a given subsegment number is called the defining SETLOC for that subsegment. ARGUS assigns the following coding to the subsegment indicated until a SETLOC is processed which specifies a different subsegment. A segment in which no subsegments are specified is assumed to consist of a single subsegment. In the case of a common subsegment, the subsegment number must be followed by the letter "C" on the defining SETLOC (the first SETLOC in any segment of the program which specifies that subsegment number). In every segment in which the common subsegment appears, it must be represented by a SETLOC which specifies the same subsegment number. (The "C" following this number is optional on all but the defining SETLOC; however, if the subsegment is not specified as common on the defining SETLOC, it must not be so specified on any SETLOC.)

The programmer may either tag a SETLOC instruction or leave the location field blank. If the instruction is tagged, the tag may be preceded by an "L" (link tag), but it may not be preceded by "F", "S", "B" (mask tag), "Z" (special register tag), or "X" (out-of-sequence tag). If the SETLOC specifies a subsegment, the command code is followed by a comma and a subsegment number from 1 to 7 (and a "C" if this is the defining SETLOC for a common subsegment).

The programmer may designate a main memory address in the A address field of a SETLOC instruction, a bank indicator in the B address field, a group indicator in the C address field, or any combination of these elements, subject to the rules stated below. If these options are not exercised, the Assembly Program assumes complete responsibility for the allocation of subsegments, guarding against overlap among subsegments and, wherever possible, against crossing a bank boundary within a subsegment. If the programmer uses the SETLOC instruction to control allocation, he must assume these responsibilities. For example, if the defining SETLOC is used to specify the initial location of a subsegment, enough room must be allowed for any mask groups, subroutines, and/or out-of-sequence words to be placed in the previous subsegment. If the programmer assumes control of allocation, he should assign an initial location which is divisible by 64 to the first subsegment in each memory bank.[1]

In a defining SETLOC, the A address field may be blank or it may contain a number up to 2047 or a symbolic tag which is equated to a number. The tag may be followed by an address modifier in the range ±16, 383, provided that the resulting subaddress is not greater than 2047. Unless the A address field is blank, ARGUS converts its contents into an 11-bit subaddress which is placed in the subaddress bit positions of the current location counter (CLC). The B address field of a defining SETLOC may be blank or it may contain a "B" followed by a number from 0 to 7 to be placed in the bank indicator bit positions of the CLC. The contents of the CLC, either modified or unmodified, specify the location of the first in-line coding word following the SETLOC.

If the defining SETLOC for a subsegment does not alter the contents of the CLC in any way (i.e., the A and B address fields are both blank), no SETLOC in that subsegment may specify a bank indicator. However, any other (non-defining) SETLOC in that subsegment may specify a main memory address in that subsegment, using a tag which is assigned to such an address or using C, ± a number. The tag may be followed by an address modifier in the range ±16, 383. If the tag has both a memory assignment and a complex assignment, the memory assignment is used. C, ±0 is equivalent to a blank or inactive A address and refers to the next available location in the subsegment. If the defining SETLOC for a subsegment does alter the contents of the CLC in any way, any other SETLOC in that subsegment may specify a main memory subaddress, a bank indicator, or both, or it may specify a tag which is assigned to a main memory address, using any of the above formats.

---

[1] See the Executive System Manual, Section II.

The C address field in any SETLOC may be blank or it may contain a "G" followed by a number from 0 to 7 which designates the special register group to be used.[1] If a group indicator is specified, the program is assembled to use the specified group and all following coding words which are marked by special register tags are loaded into that group. If the C address field is blank, the previous group specification remains in effect. If no group has been previously specified within the same segment, the Assembly Program uses group 1. As noted in Section VI, it is the programmer's responsibility to specify a group in which registers S4 through S7 are available if his program uses those registers. (Note that these registers are normally unavailable in group 1.) The programmer may use as many special register groups as he requires and may change groups as often as necessary.

EVEN

Each special register group includes an unprogrammed transfer register (UTR), which should be set up with the initial address of a group of instructions to handle the various unprogrammed transfer conditions described in Appendix E. This initial address must be an even number for proper execution of the unprogrammed transfers. The Assembly Program assigns the next even-numbered address in sequence to the word following the EVEN instruction. The programmer should write a symbolic tag in the location field of either the EVEN instruction or the following word. This tag may be a link tag, but it may not be a mask tag, a special register tag, or an out-of-sequence tag. The special address constant (see Section IX) may be used to load the address assigned to this tag into the UTR. The three address fields are not used in the EVEN instruction.

It is the programmer's responsibility to set up the UTR and to provide enough instructions following EVEN to provide for any unprogrammed transfer situations which may arise in his program. He may use SETLOC, MODLOC (below), or any other valid method in place of EVEN to assure the assignment of an even address for loading the UTR.

SIMULATE

Every simulator routine is preceded by the instruction SIMULATE, which is punched with a symbolic tag in the location field to identify the routine. The three address fields are not used in the SIMULATE instruction. The tag of a SIMULATE instruction may be a link tag, but not a mask tag, special register tag, or out-of-sequence tag. The Assembly Program assigns this tag to the next location in sequence which has three binary ones (octal 7) in its low-order subaddress bits. The first word of the simulator routine is then assigned to

[1] Note that a program assembled to use group 0 may not run properly under control of the Program Test System.

the following location. To set up and perform the routine, the tag of the SIMULATE instruction is referenced in the command code field of a simulator instruction, as described in Section VII.

MODLOC

This instruction directs the Assembly Program to allocate the following word to the next location whose address is a multiple of 2, 4, 8, 16, 32, or 64, as specified by the number punched in the A address field. The B and C address fields are not used. Any tag written in the location field of the MODLOC instruction, or of the following word, is assigned to the address of the location to which the following word is allocated. This tag may be a link tag, but not a mask tag, special register tag, or out-of-sequence tag. Note that the address to which the following word is assigned always ends in from one to six binary zeros, depending upon the number specified in the A address field.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	REMARKS 66 LINE NUMBER 73	74 80
SELTYPE	MODLOC		8				

This instruction causes the Assembly Program to allocate the following word to the next location in sequence whose address is a multiple of 8 and to assign the tag SELTYPE to the address of this location.

ASSIGN

This instruction assigns a tag to a complex address, such as an indexed or an indirect address. The programmer writes the tag to be assigned in the location field and the complex address in the A address field. The B and C address fields are not used. The tag in the location field may not be a link tag, a special register tag, a mask tag, or an out-of-sequence tag; however, it may be assigned elsewhere in the program to a memory location address (as described on page 20). The use of the ASSIGN instruction allows the programmer to change item formats and to reassign special registers during reassembly, changing only the ASSIGN instructions rather than changing every reference to the corresponding addresses. For example, to assign the tag GROSSPAY to indexed address 3, 5, and the tag PRODUCT to indirect address N, R3, 12, the programmer writes the following two instructions.

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	REMARKS 66 LINE NUMBER 73	74 80
1	GROSSPAY	ASSIGN		3, 5				
2								
3	PRODUCT	ASSIGN		N, R3, 12				

TAS (Temporary Assignment)

This instruction also assigns a tag to a complex address. However, a tag which has been assigned by means of a TAS instruction may be freely reassigned to another complex address by means of another TAS. The instruction is written in the same format as ASSIGN and the same rules apply to the types of tags that may be assigned. The programmer may use the TAS instruction to reference the same set of data by several different complex addresses, using only a single tag. In the following example, the tag DATA is first assigned to the indexed address 3, 0, then later reassigned to the indirect address N, R1, 1.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C 24	A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	66 REMARKS LINE NUMBER 73	74 80
1	DATA	TAS		3,0				
2								
3	DATA	TAS		N, R1, 1				

EQUALS

The EQUALS instruction assigns a value to a symbolic tag. The assigned value may be an integer, another symbolic tag, or an expression which is an algebraic combination of up to six integers and tags. Addition (+), subtraction (-), multiplication (*), and division (/) may be used to combine integers and symbols. These operations are performed in the following order:

1. Multiplication and division;
2. Addition and subtraction.

Parentheses are not permitted in the combination of integers and symbols. This instruction is useful in combining programs which use different symbols and in altering such parameters as data block lengths.

The tag to be equated is written in the location field and must not be a link tag, mask tag, special register tag, or out-of-sequence tag. The equated value is written starting in the A address field and continuing through as many consecutive columns as necessary. Any symbol used in an equated expression must have been previously assigned either to a memory location address or to an integer. The computed value of the expression must be either a valid memory address or an integer. If the only term in the equated expression is a tag which has both an absolute and a complex assignment, then the same two values are assigned to the tag in the location field. However, if such a tag is used in an algebraic combination of terms, its absolute value is used to compute the expression and its complex value is ignored.

Care must be taken in combining symbols which are assigned to memory addresses, since some combinations are meaningless (e.g., the product of two memory addresses). The programmer may determine whether or not a given combination of symbols is meaningful by examining it in terms of "dimension", which is defined as follows: the dimension of an integer or of a symbol assigned to an integer is defined to be 0, while the dimension of a symbol assigned to a memory location is defined to be 1. The dimension of an equated expression must be 0 or 1, where:

1. The dimension of the sum (difference) of two terms equals the sum (difference) of their dimensions; and
2. The dimensions of the two factors in a product or quotient must both be 0.

In addition to these dimensional requirements, symbols which are combined must have been assigned in the same subsegment and by the same location counter (CLC or XLC). A mask tag is not permitted in an equated expression unless it is the only term in the expression.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION 1–10	COMMAND CODE 11–22	S/C 23	A ADDRESS 24–37	B ADDRESS 38–51	C ADDRESS 52–65	REMARKS 66–80
1	ENDMATRX	EQUALS		MATRIX+M*N-1			
2							
3	PAYROLL	EQUALS		PROLL			
4							
5	ALENGTH	EQUALS		APRODUCT-BPRODUCT			

The first instruction above assigns the tag ENDMATRX to the final location of an M by N matrix whose initial location is tagged MATRIX. Since M and N are integers, the dimension of the entire expression is 1. The second instruction equates the tag PAYROLL to the tag PROLL, which might be used to represent the same quantity in another program. The third instruction equates the tag ALENGTH to the difference of two memory locations, which is the length of a table and has a dimension of 0. As stated above, the tags APRODUCT and BPRODUCT must have been assigned in the same subsegment and by the same location counter.

RESERVE

This instruction is used to reserve a block of memory locations for data or working storage. The number of locations to be reserved is specified by means of an integer, a tag, or a combination of integers and tags which starts in the A address field and continues through as many consecutive columns as necessary. The same rules apply for combining integers and tags as in the EQUALS instruction (above), except that the dimension of the combination must be 0. Any tag which appears in the combination must have been

previously assigned to an absolute value (memory location address or integer). If such a tag has an additional complex assignment, this assignment is ignored and the absolute assignment is used in computing the value of the combination. If the programmer writes a tag in the location field, it is assigned to the first reserved location. This may be a link tag or an out-of-sequence tag, but not a mask tag or a special register tag. For example, the first instruction below reserves 100 locations starting at the location tagged INPUT. The second instruction reserves M times N locations starting at the location tagged MATRIX (where M and N have been previously assigned integer values).

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C 24	A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	66 REMARKS LINE NUMBER 73	74 80
1	INPUT	RESERVE		100				
2								
3	MATRIX	RESERVE		M * N				

MASKGRP

Before any masks can be designated, generated, or referenced in a segment, the control instruction MASKGRP must be written to assign a shift group number, a field group number, or both. The only exception is at the beginning of a segment, where any designated or generated masks are automatically assigned to field and shift groups 0 if they are not preceded by a MASKGRP instruction.

The location field is not used in a MASKGRP instruction. The A address field of this instruction may specify the group number of a shift mask group (an "S" followed by a comma and a number from 0 to 15), and the B address field may designate the group number of a field mask group (an "F" followed by a comma and a number from 0 to 15). The operation code may be followed by a comma and the identification number of the subsegment in which the specified mask groups are to be stored. If the subsegment number is omitted, the specified mask groups are stored in the subsegment which is in control at the end of the segment. Up to 16 field mask groups and 16 shift mask groups may be set up within any segment. However, any groups which are stored in a common subsegment are included in the total number of groups for every segment of the program. A shift group and a field group having the same group number (e.g., S, 2 and F, 2) must be stored in the same subsegment.

A MASKGRP instruction directs that all of the following designated or generated masks belong to the groups specified until another MASKGRP specifies different groups. The Assembly Program assigns a mask base address to each specified group. The base of a group of field masks must be a multiple of 32; that of a group of shift masks must be a

multiple of 64. Each designated or generated field or shift mask is assigned the next sequential location within the proper group until either the group is full or a new group of the same type is specified. Any mask assigned with a mask indicator of "B" (for use with both shift and field instructions) must be preceded by a MASKGRP instruction in which the group numbers are equal. When a "B" mask is assembled, an overlapping pair of mask groups is set up which can include up to 32 field, shift, or "B" mask and up to 32 additional shift masks.

The MASKGRP instruction also directs that all following mask references are to masks in the specified groups until different groups are specified. Proper execution of a shift instruction or a masked field instruction requires that the mask index register be set up with the base address of the desired mask group. This is done by loading or transferring a MASKBASE constant (see Section IX) into the mask index register. Since machine instructions can only reference masks in the current groups, any reference to a mask in another group must be preceded by both a new MASKGRP instruction and the necessary coding to change the mask index register setting.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION 1–10	COMMAND CODE 11–22	S/C 23	A ADDRESS 24–37	B ADDRESS 38–51	C ADDRESS 52–65	REMARKS 66–80 (LINE NUMBER 73)
1		MASKGRP,2		S,1	F,1		
2							
3		MASKGRP,2			F,2		
4							
5		MASKGRP		S,4	F,3		

The first MASKGRP instruction above designates that all following shift masks are in shift group 1 and all field masks in field group 1 until the next MASKGRP is processed. Furthermore, these mask groups are to be stored in subsegment 2. If any "B" masks appear in these groups, the groups will be overlapping; otherwise, storage will be provided for the full 96 masks if required. Should the entire field mask group be assigned while space remains for additional shift masks, for example, the second MASKGRP instruction can be written to set up field mask group 2 in subsegment 2. Any reference to a field mask in group 1 after this second group is designated must be preceded by a MASKGRP instruction redesignating field group 1. The third instruction above may be written at a later point in the program to designate shift group 4 and field group 3. Since no subsegment number is written, these groups will be stored in the subsegment which is in control at the end of the segment. The latter groups may not include any "B" masks as their group numbers are not identical.

END

Every program being assembled should include an END card, though the position of this card in the program deck is irrelevant. The information punched on the END card is provided for use by the Executive System. The command code is followed by a comma and the number of the special register group to be given control at the start of the program.[1] If no group is specified, control is given to group 1. The program name is punched in the A address field and the name of the first segment to be loaded is punched in the B address field. The location and C address fields are not used. An existing program does not require a new END card for reassembly unless any of the information on the original END card is to be changed. When a program is loaded by Executive, the segment named is loaded first. This segment must contain coding to load the sequence counter of the group specified as first in control.

[1] Note that a program which gives initial control to group 0 may not operate properly under control of the Program Test System.

SECTION IX

CONSTANTS

Data constants in a number of different formats can be assembled. In addition, Assembly recognizes several control constants which are used for special functions. Each type of constant is identified by a constant code punched in the command code field.

Data Constants

The seven types of data constants recognized by Assembly are:

ALF	alphanumeric	FXBIN	fixed-point binary
OCT	octal	FLBIN	floating-point binary
DEC	fixed-point decimal	FLDEC	floating-point decimal
		EBC	extended binary

Several constants of the same type may be combined on a single card, the maximum number depending on the type of constant. Alphanumeric, octal, and fixed decimal constants are specified in the desired notation. All of the other constants are specified in fixed decimal notation and translated by Assembly. The constants are punched starting in the A address field and continuing through as many consecutive columns as necessary (up to column 65). All data constants except alphanumeric are separated by commas and may include spaces to aid in visual checking; these spaces are ignored by Assembly.

The location field of a card which contains a single data constant may be blank or it may contain any standard tag configuration (see Section IV). The location field of a card which contains more than one constant may be blank or it may contain X, or a symbolic tag, or X, followed by a symbolic tag. Any tag written in this field references the first constant on the card. All constants punched on the same card are allocated to consecutive memory locations, under control of the designated location counter, so that they may be referenced by address arithmetic.

ALF (Alphanumeric Constant)

If the constant code (ALF) is followed by a comma and a number from 1 through 5, ARGUS assembles the indicated number of eight-character constants from the contents of columns 24 through 31, 32 through 39, 40 through 47, etc. Spaces are valid characters and are assembled by ARGUS. Sentences or related words may be punched consecutively across a card through column 63. If no information follows ALF in the command code

field, n is assumed to be 1 and a single constant is assembled from the contents of columns 24 through 31.

Alternatively, ALF may be followed by a comma and any character other than 0 through 5, which is then interpreted as a terminating character but is not assembled. This character is repeated following the last word punched and should not appear in any of the punched constants (see second example below).

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION (1–10)	COMMAND CODE (11–22)	S/C	A ADDRESS (24–37)	B ADDRESS (38–51)	C ADDRESS (52–65)	REMARKS (66–80)
1	ALPHA 1	ALF, 4		NEVER BEFORE	IN HISTORY	HAVE SO	
2							
3		ALF, %		MANY OWED SO	MUCH TO SO	FEW %	
4							
5	ALPHA 2	ALF, 5		A B	C D	E	

OCT (Octal Constant)

Up to 16 unsigned or 15 signed octal digits per word may be punched starting in column 24 and continuing through column 65. If 15 signed digits are specified, the most significant digit must be less than 4 (as shown in Figure 2, page 8). Words are separated by commas and stored in consecutive memory locations. Signed words are justified to the right, i.e., the least significant digit is placed in digit position 16, the sign is placed in position 1 (0000 if negative, 1111 if positive), and positions between the sign and the most significant digit are filled with zeros. Unsigned words are justified to the left; i.e., the most significant digit is placed in digit position 1 and positions following the least significant digit are filled with zeros.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION (1–10)	COMMAND CODE (11–22)	S/C	A ADDRESS (24–37)	B ADDRESS (38–51)	C ADDRESS (52–65)	REMARKS (66–80)
1	OCTCON 1	OCT		01234567765432	10, 765432 10012	34567	
2							
3	OCTCON 2	OCT		+175, -2, 324, +3	24150416723454		

DEC (Fixed Decimal Constant)

Decimal constants are punched from column 24 up to column 65 and are separated by commas. Each word may contain up to 11 signed or 12 unsigned decimal digits. Any hexadecimal digit (0-9 and B-G) may be specified. Unsigned decimal constants are justified left and signed decimal constants are justified right. If the programmer wishes to position a constant other than by the above rules, he may follow the constant with "P"

and a number from 1 to 12 specifying the storage position of the units digit. A decimal point may be written to indicate the units digit; otherwise, the low-order digit is assumed to be the units digit. If a decimal point is written, the programmer must specify the position of the units digit.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	REMARKS 66 LINE NUMBER 73 74 80
1	DECCON 1	DEC		123456789, +12,	-34, 123.516P7,	789BCDE, 4G4G	
2							
3		DEC		2			

FXBIN (Decimal to Fixed Binary Translation)

Fixed-point binary constants to be converted by Assembly are specified by their decimal equivalent. The constants, separated by commas, are punched starting in column 24 and are stored in consecutive locations in memory. Each constant may contain up to 14 decimal digits, a decimal point, and a sign. If no sign is specified, a plus sign is assumed. If a decimal point is specified, the bit position of the units bit must be designated by a "B" and a number from 4 to 48 immediately following the constant. If there is no decimal point, the constant may have any absolute value up to $2^{44} - 1$ (or 17, 592, 186, 044, 415). If the decimal point is at the far left, the constant is accurate up to $\pm 2^{-44}$; i. e., it may contain up to 14 significant digits. If the decimal point is anywhere else, its position determines the maximum value of the constant.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	REMARKS 66 LINE NUMBER 73 74 80
1	FXBCON 1	FXBIN		+3, +24, 256. 125B13,	-32.832832B18		

FLDEC (Floating-Point Decimal Constant)

Floating-point decimal constants are punched starting in column 24 and separated by commas. They may be specified with a decimal point, an explicit exponent, or both. An explicit exponent consists of an "E" and a signed or unsigned exponent immediately following the constant. However such constants are specified, they are converted to floating-point form and normalized by Assembly. Each constant may contain up to 10 signed decimal digits and may range in value from 10^{-65} virtually to 10^{63}. Unsigned constants and exponents are considered positive. The structure of a floating-point constant is shown in Figure 2, page 8.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

LOCATION	COMMAND CODE	S/C	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
FLOCON 1	FLDEC		+123.456E+15,-12489E-20,14856.12389			

When the three sample constants above are converted and normalized, they will assume the following forms:

+123.456E+15	becomes	+.123456 times 10^{18}
-12489E-20	becomes	-.12489 times 10^{-15}
14856.12389	becomes	+.1485612389 times 10^{5}

FLBIN (Floating-Point Binary Constant)

These constants are punched in the same manner as floating-point decimal constants. They are converted to floating-point binary form and normalized by Assembly. Each constant may contain up to 13 signed decimal digits and may range in value approximately from 10^{-78} to 10^{76} (16^{-65} to 16^{63}). This is equivalent to the range of 2^{-260} to 2^{252}. In floating-point binary form, the exponent represents a power of 16. Unsigned constants are considered positive.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

LOCATION	COMMAND CODE	S/C	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
FLBCON	FLBIN		+123.456E+15,-12489E-20,14856.12389			

EBC (Extended Binary Constant)

Extended binary constants are punched in the same format and have the same range of values as floating-point binary constants. Each constant may contain up to 25 signed decimal digits. These constants are converted into normalized, double-precision, floating-point binary numbers, retaining 80 binary digits of the mantissa. Assembly stores the high-order 40 bits, with proper exponent and sign, as one machine word. It stores the low-order 40 bits, with the same sign and an exponent 10 less than that of the high-order word, as the following word.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

LOCATION	COMMAND CODE	S/C	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
EBCON	EBC		+123.456E+15,-12489E-20,14856.12389			

Control Constants

The Assembly Program recognizes the following control constants:

SPEC	special address	M	mixed
CAC	complete address	TAC	tape address
MASKBASE	mask base	LINK	linkage
CONTROL	program control	SEGNAME	segment name
		SUBCALL	subroutine call

These constants are punched one to a card.

SPEC (Special Address Constant)

A special address constant specifies an address to be stored in special register format. If the location field contains a special register tag (see page 18), this constant is loaded directly into the designated register. Otherwise, it is allocated normally in memory in the proper form for transfer to a special register (or for comparison with the contents of a special register). As already noted, a special register has the capacity to store a sign, a bank or group indicator, and a subaddress.

The A and B address fields are not used in a special address constant. The sign of the specified address may be written in the S/C column. If no sign is included, a positive sign is assumed. If no information follows the constant code, the contents of the C address field are interpreted as a complete address which may be specified in several different ways:

1. An integer up to 16, 383;
2. A symbolic tag which has an absolute assignment, with or without an address modifier in the range ±16, 383. If the tag has both an absolute and a complex assignment, the absolute assignment is used;
3. C, with or without an address modifier in the range ±16, 383. C, is replaced by the current contents of the CLC;
4. X, with or without an address modifier in the range ±16, 383. X, is replaced by the current contents of the XLC; or
5. Blank which is replaced by a bank indicator and subaddress of all zeros.

If the constant code is followed by a comma, a "G", and a group indicator (0-7), the C address field is interpreted as a special register address which may be specified in several different ways:

1. An integer up to 2047;
2. A symbolic tag which is equated to an integer, with or without an address modifier in the range ±16, 383, provided that the result does not exceed 2047;
3. A direct (absolute or mnemonic) special register address or an indirect address;

4. A tag which has a special register assignment or a complex assignment; or
5. Blank which is replaced by a special register address of all zeros.

Any reference to the read-write address counters associated with a tape or peripheral control unit must be relocated independently of other references to special register groups. These counters are addressed using special address constants with indexed special register addresses (see page 25). The constant code (SPEC) is followed by a comma and the control unit indicator (A-H) corresponding to the desired counter. The C address field may contain any of the above formats, but should result in a configuration that can be used successfully as the index register contents in an indexed special register address. Assembly combines the subaddress portion of this information with the group indicator which corresponds to the specified control unit indicator to form a complete address. Normally, if a SPEC constant designates a control unit indicator, the C address field contains either the direct address of the desired counter, an indirect address which references the desired counter, or 0. When a SPEC constant of this type is stored in an index register, it can be used with an indexed special register address of the proper type to address the desired read-write counter.

The special address constant is also used to set up the stopper address in special register format, as mentioned in Sections V and VI. In this case, the constant is written with the symbolic tag STOPPER in the C address field and no information following the constant code.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION (1–10)	COMMAND CODE (11–22)	S/C (23)	A ADDRESS (24–37)	B ADDRESS (38–51)	C ADDRESS (52–65)	REMARKS (66–80)
1	Z,R2	SPEC				PAYROLL	
2		SPEC,G5				Z,CSC	
3	Z,X1	SPEC				STOPPER	
4		SPEC,A				0	

The first special address constant above loads special register R2 of the controlling group with the memory location address assigned to the tag PAYROLL. This tag must appear in the location field of another card. The second constant stores the address of the cosequence counter of group 5 in special register form. The third stores the complete address of the stopper location in index register 1 of the controlling group. The last constant stores a complete address consisting of a plus sign, the group indicator which corresponds to control unit indicator A, and a subaddress of all zeros.

CAC (Complete Address Constant)

A complete address constant specifies up to three complete addresses to be stored in one memory location. The constant consists of three 16-bit groups, each containing a sign, bank indicator, and subaddress in special register format. It is used to store addresses which are to be transferred to special registers, but it may not be compared directly with the contents of a special register, unless the left-most two addresses are all zeros.

The three addresses which are to be stored in the left-most, middle, and right-most groups are punched in the A, B, and C address fields, respectively. Each address may be specified as an integer (up to 16, 383) or as a symbolic tag, with or without an address modifier in the range ±16, 383. If a tag is written, it must have an absolute assignment. If it has an additional complex assignment, the absolute assignment is used. If any field is left blank, the corresponding group will contain all zeros. The S/C column may contain a sign to be included in all of the three addresses which are not blank. If no sign is written, a plus sign is assumed.

Note that the address written in the A address field can be stored in a special register by means of a 32-bit shift to the right, the address in the B address field by means of a 16-bit shift to the right, and that in the C address field by means of a transfer instruction.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	REMARKS 66 LINE NUMBER 73 74 80
1		CAC		PAYROLL	4678	HOURS-653	
2		CAC	-		INVENTRY	TOOLS	

The first constant results in a word containing, from left to right, the complete address assigned to the tag PAYROLL, the absolute address 4678 and the address 653 before that assigned to the tag HOURS. All three addresses have positive signs. The second constant results in a word containing, from left to right, an address of all zeros and the addresses assigned to the tags INVENTRY and TOOLS. The addresses in the B and C address fields are combined with negative signs.

MASKBASE (Mask Base Address Constant)

A single setting of the mask index register (MXR) stores two mask base addresses: the base of a group of up to 32 field masks and the base of a group of up to 64 shift masks. These two addresses must be in the same bank, as the mask index register contains only one bank indicator which is used with both bases.

The MASKBASE constant contains a shift group number in the A address field and a field group number in the B address field. The C address field and the S/C column are not used. This constant results in a special register word containing the base addresses of the two groups specified, in the format required by the mask index register. The safest way to insure that both base addresses are in the same memory bank is to write the numbers of two groups which are specified by the same MASKGRP instruction, or which are at least in the same subsegment.

If the MASKBASE constant is tagged Z, MXR in its location field, the mask index register will be loaded directly. Otherwise, the MXR must be loaded by means of a programmed transfer. Any reference to a mask in a given group must be preceded by the coding which sets up the MXR with the base address of that group. If the programmer desires to change only the shift mask base in the MXR, for example, he may load a MASKBASE constant containing the new shift group number and repeating the previously loaded field group number.

In order to give the programmer more control over the allocation of masks, it is possible under certain conditions to set up a mask group without using the MASKGRP instruction and without designating the masks by mask tags. It is the programmer's responsibility to insure that the masks in such groups are stored in consecutive locations and that the rules governing mask base addresses are met. (The SETLOC or MODLOC instruction can be used to comply with these modular restrictions.) Mask groups which are set up in this manner can be referenced by loading the MXR with a MASKBASE constant which contains the tag of the shift mask base in the A address field and the tag of the field mask base in the B address field. The masks themselves may be tagged (without mask indicators) and referenced symbolically in shift and field instructions in the normal manner. These tags must have absolute assignments; if they also have complex assignments, the absolute assignments will be used. Address arithmetic may be used in referencing masks of this type, as well as in the A and B address fields of the MASKBASE constant. Moreover, this constant may specify one mask group symbolically and the other by group number, provided that both are in the same memory bank. This method of setting up mask groups may not be used in any segment which includes subroutines, macro routines, or generated masks. Under any of these conditions, the MASKGRP instruction must be used.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION (1–10)	COMMAND CODE (11–22)	S/C (24)	A ADDRESS (24–37)	B ADDRESS (38–51)	C ADDRESS (52–65)	REMARKS (66–80)
1	Z,MXR	MASKBASE		S,1	F,1		
2		MASKBASE		S,1	F,2		
3		TX		C,-1		Z,MXR	
4	Z,MXR	MASK BASE		SBASE	F BASE		

The first constant above results in a special register word containing the base addresses of shift group 1 and field group 1 in MXR format, which is loaded directly into the mask index register. Later, if it is desired to set up the MXR so that the program can reference masks in field group 2, continuing to reference shift group 1, the second constant may be transferred into the MXR by means of a TX instruction, as shown. The last constant illustrates the special use of the MASKBASE constant when the programmer assigns the mask group bases and no MASKGRP instruction is used.

CONTROL (Program Control Constant)

A program control constant may be used as a mask for examining the contents of the program control register (PCR). If the constant is to be used for examining the program demand bits, the bisequence bits, or both, no information follows the constant code. The group indicators of up to seven control groups whose program demand bits are to be examined are written in the A address field, separated by commas. The group indicators of up to seven groups whose bisequence bits are to be examined are written in the B address field, also separated by commas. The resulting constant will contain binary ones in the bit positions corresponding to the specified positions of the PCR.

If the program control constant is to be used for examining the buffer interlock bits, the constant code is followed by a comma and a "B" (for buffer). The indicators (A-H) of up to seven control units whose input buffer interlock bits are to be examined are written in the A address field; those of up to seven control units whose output buffer interlock bits are to be examined are written in the B address field, all separated by commas. The resulting constant will contain binary ones in the bit positions corresponding to the specified positions of the PCR.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION 1–10	COMMAND CODE 11–22	S/C	A ADDRESS 24–37	B ADDRESS 38–51	C ADDRESS 52–65	REMARKS LINE NUMBER 66–73	REMARKS 74–80
1		CONTROL		1,2	2,7			
2								
3		CONTROL,B		A,C,D	E,G			

The first program control constant above generates a mask to examine program demand bits 1 and 2 and bisequence bits 2 and 7. The second generates a mask to examine the input buffer interlock bits for control units A, C, and D, and the output buffer interlock bits for control units E and G. The contents of the PCR must be transferred to a memory location by means of a program control instruction (see page 50) before they can be examined.

M (Mixed Constant)

A mixed constant contains four fields. The constant may include octal, decimal, or alphanumeric characters or any valid address format, but each field may contain only one type of character or one address. "M" is written in the command code field followed by a comma. The remainder of the command code field and the three address fields correspond, respectively, to the four 12-bit groups in a Honeywell 800 word (see Figure 2, page 8). Each of these fields may contain one of the following:

1. An "A", a comma, and two alphanumeric characters;
2. A "B", a comma, and four octal characters;
3. A "D", a comma, and three hexadecimal characters; or
4. Any valid address format described in Section V.

An exception is the command code field, which may not contain a special register address. An "S" or a "C" in the S/C column results in a 0 bit or a 1 bit, respectively, in bit position one, overriding whatever the constant puts in this bit position. A blank S/C column is ignored. Note that if one or more address fields contain special register addresses, the left-most 12-bit group may not contain the configuration indicated in the command code field.

A mixed constant can be used to store one or more addresses for use in setting up a program in memory. If a mixed constant contains no symbolic tags, it is actually a data constant and can be used to specify a data word in compressed alphanumeric form.

The first mixed constant in the following example is stored in memory as decimal 009, followed by the subaddress assigned to the tag AB+5, followed by 24 binary ones.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION	COMMAND CODE	S/C	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1		M,D,009		AB+5	B,7777	B,7777	
2		M,INPUT 1		INPUT 2	D,586	OUTPUT 1	
3		M,A,JK		A,LM	GROSSPAY	0,37	

TAC (Tape Address Constant)

This constant is used to specify up to eight tape or peripheral codes to be stored in one memory location. The codes (AA-HH) are written starting in the A address field, separated by commas, and continuing through as many consecutive columns as necessary. The resulting machine word contains the corresponding six-bit peripheral addresses justified to the left. Any unspecified codes to the right of the last code written are filled with binary ones. Any unspecified codes to the left of the last code written should be specified as GG.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION	COMMAND CODE	S/C	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1		TAC		AA,AD,BA,GG,GG,HA,GC			

LINK (Linkage Constant)

The linkage constant is used in the Executive macro routine read segment (see page 30). The C address field contains the link tag which the programmer writes in the B address field of the read segment instruction. Address arithmetic is not permitted. The A and B address fields are not used. The link tag must have a memory location assignment; if it has an additional complex assignment, the memory location assignment is used. The linkage constant is interpreted by Executive, and only by Executive, as the starting location of the next segment to be loaded. It is loaded by Executive in special register format, with the assigned value of the tag in the bank indicator and subaddress positions. The programmer is not required to write linkage constants unless he is writing additional Executive macro routines.

SEGNAME (Segment Name Constant)

The segment name constant is used in the sort routine calling sequence. It is an alphabetic constant containing the name of the segment in which it appears. The programmer is not required to write segment name constants unless he is writing a macro routine for use as a sort routine calling sequence.

SUBCALL (Subroutine Call Constant)

The subroutine call constant is used in all macro routines which serve as calling sequences for library subroutines (see Section XIII). ARGUS replaces this constant with a special address constant containing the address of the subroutine entry. The programmer is not required to write subroutine call constants unless he is writing a macro routine for use as a subroutine calling sequence.

SECTION X

MASKING

Many of the preceding sections have discussed masking since this subject relates to most aspects of the assembly language. As a result, the references required for a firm understanding of masking are found in widely scattered points throughout the manual. The present section summarizes and illustrates the specification and use of masks in assembly notation.

If an instruction in a program is to operate on part of a word while ignoring the rest, that instruction must use a word called a mask. This is a 48-bit word containing binary ones in the bit positions which the instruction is to examine or use and zeros elsewhere. Instructions which use masks have, in effect, a fourth address for the purpose of referencing the mask.

Many of the machine instructions in the general group can be performed under the control of masks. Such instructions are called field instructions. In addition, all of the shift instructions require the use of masks in their execution. Masks may be specified by the programmer, stored in memory with the proper mask tag, and referenced in the command code field of either a field or a shift instruction.

Designated Masks

If the word +58393857320 is stored in memory and the programmer wishes to work only with the low-order four digits, he may write a mask which contains binary ones in the low-order 16 bit positions. Such a mask could be written as a decimal constant and tagged, for example, MASK1. The tag should be preceded by S, F, or B, to designate whether it is to be used with shift instructions, field instructions, or both.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

1 LOCATION 10	11 COMMAND CODE 22	23 S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	66 REMARKS LINE NUMBER 73	74 80
B,MASK 1	DEC		000000006666				

This is a mask containing 16 low-order 1 bits to be used with both shift and field instructions. Reference to this mask could be made in the command code field, where the tag MASK1 would be separated from the operation code by a comma, or in any of the address fields. When the mask is referenced in the command code field, it is used as a mask, and when it is referenced in an address field, it is used as an operand. [1]

[1] Exceptions are the general instructions substitute (SS) and extract (EX), which always reference a mask in the B address field. However, since the mask is not referenced via the mask index register, it need not have a mask tag nor be stored as part of a mask group.

Generated Masks

Alternatively, in any field or shift instruction, the programmer may specify information from which ARGUS can generate masks during assembly. This information includes the number and type of characters in the masked field, the position of the left-most character in the field, and the position(s) of any sign bit(s) which is attached to the field. The use of generated masks is limited to alphanumeric or hexadecimal fields of consecutive characters.

To generate a decimal mask for the low-order four digits and the sign of the word +58393857320 the programmer would write the command code in this manner:

TSD, 9, 4, S

This tells Assembly to generate a mask containing ones in the bit positions to be examined and to put the relative address of this mask in the operation code. When this instruction is executed under control of the generated mask, hexadecimal digits 9 through 12 and the sign will be transferred from the location specified in the A address field to the location specified in the B address field. A generated mask may be used with a field instruction only if the type of characters in the masked field is inherent in the operation code. (See page 40 for further discussion of generated masks.)

Mask Groups

Masks are stored in memory in groups. Machine instructions reference masks relative to an address called the base address of a mask group. A field mask group may contain up to 32 masks and must have a base address which is a multiple of 32. A shift mask group may contain up to 64 masks and must have a base address which is a multiple of 64. The programmer may group his masks in consecutive locations and assign valid base addresses (using SETLOC or MODLOC) under certain circumstances. When masks are allocated by the programmer they need not be marked by mask tags. Alternatively, the programmer may direct Assembly to allocate mask groups by using the control instruction MASKGRP. In any segment which contains subroutines, macro routines, or generated masks, Assembly must control the allocation of mask groups.

If the MASKGRP instruction is used, it must appear before any masks are generated, designated, or referenced within a segment. The only exception is at the beginning of a segment, where any designated or generated masks are automatically assigned to field and shift groups 0 if they are not preceded by a MASKGRP instruction. This instruction may include a shift group number, a field group number, or both. It directs Assembly:

1. To assign a valid base address to each specified group;
2. To allocate all subsequent designated or generated masks to the proper

specified group until another MASKGRP instruction specifies a group of the same type; and

3. To obtain all masks referenced from the proper specified group until another MASKGRP instruction specifies a group of the same type.

The base address of a mask group is not necessarily the starting address of that group. When the first mask in a group is processed, Assembly attempts to assign the last previous address of the proper modular value (32 or 64) as the base address of that group. This involves counting the masks in the group to determine whether they can all be allocated before another address of the same modular value is reached. If this can be done, the base address is assigned as above. If not, the next address of the proper modular value is assigned as the base address of the group.

The MASKGRP instruction may also specify the subsegment in which the mask group(s) is to be stored. If no subsegment is specified, the masks are stored in the subsegment which is in control at the end of the segment. A pair of groups having the same group number must be stored in the same subsegment. When masks are allocated by Assembly, each designated mask must be so marked by preceding it with a mask tag.

If any type B masks are designated, Assembly sets up overlapping shift and field mask groups. A pair of overlapping groups can store 32 field, shift, or both masks, plus an additional 32 shift masks. If mask groups are allocated by Assembly, the first B mask must be preceded by a MASKGRP instruction which specifies identical shift and field group numbers. If they are allocated by the programmer, the first B mask must be preceded by the necessary control instructions to set up overlapping groups.

Referencing Masks

Both shift and field instructions reference masks by means of a special register called the mask index register (MXR). This register stores the base subaddress of a shift group, the base subaddress of a field group, and a bank indicator to be used with both subaddresses, all according to a special format. Before any mask in a given group can be referenced, the base address of that group must be stored in the MXR by means of the control constant MASKBASE. If mask groups are allocated by Assembly, the MASKBASE constant is written with the group numbers of the desired shift and field groups. If mask groups are allocated by the programmer, this constant is written with the tags which represent the base addresses of the desired groups. In either case, the MASKBASE constant must specify both a shift group and a field group. If only one subaddress is to be changed in the MXR, the current group of the opposite type is

respecified. The MASKBASE constant may be loaded directly into the MXR or moved there by a programmed transfer. Since the MXR contains only one bank indicator, shift and field mask groups which are referenced concurrently must be stored in the same memory bank.

Note that before a mask in a given group can be referenced, it is necessary to set up the MXR with the base of that group. If MASKGRP instructions are used, it is also necessary to specify the desired group in a MASKGRP instruction. The control instruction MASKGRP should not be confused with the MASKBASE constant. The MASKGRP instruction directs Assembly in allocating mask groups and assigning their base addresses. The MASKBASE constant, on the other hand, stores two mask base addresses in the special MXR format.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	66 REMARKS LINE NUMBER 73	74 80
1		SETLOC, 2						
2		MASKGRP		S, 1	F, 1			
3		TX		AB		Z, MXR		
4		DA, M1		A	B	C		
5		SWE, M3		D	12	D		
6								
7	F, M1	DEC		000000GGGGGG				
8	S, M3	DEC		00GG00GG0000				
9	AB	MASKBASE		S, 1	F, 1			

Figure 9. Designation and Referencing of Masks

EXAMPLE: The coding shown in Figure 9 illustrates the designation of masks and their use in both shift and field instructions. This coding represents only a portion of a segment. The first instruction directs Assembly to begin allocation of subsegment 2. The MASKGRP instruction assigns the following designated masks to field group 1 or shift group 1 and tells Assembly that the following instructions may reference masks from these groups. A one-word transfer is performed to load the MXR. This is followed by two program instructions which utilize field mask M1 and shift mask M3, respectively. Masks M1 and M3 are designated at another point in the coding, together with the MASKBASE constant which is used to set up the MXR with the base addresses of field group 1 and shift group 1.

Subroutine and Macrocoding Masks

The use of masks in subroutines and in macro routines (see Section XIII) varies with the type of routine. The following cases may be distinguished:

1. Macro routines in the library may contain their own masks. If a macro routine is called in a segment, any masks which it contains are automatically stored in the current groups of main program masks and must be considered in figuring the storage total of these groups. Their positions within the mask groups depend upon the position of the macro instruction relative to the main program masks.
2. Dependent subroutines are similar to library macro routines in that they may contain masks which are included in the current groups of main program masks.
3. Independent subroutines contain all the necessary coding to designate their own masks, set up the mask index register, and restore this register before returning control to the main program.

Mask Pools

To save memory space by eliminating duplicate masks, certain of the masks belonging to a group may be collected into a mask pool during assembly. All generated masks, macro routine masks, and masks appearing in dependent subroutines, which are assigned to a specific field or shift mask group, are automatically included in the pool for that group, and duplicates among these masks are eliminated. A designated mask is included in the pool of the group to which it is assigned if a "C" is written in the S/C column and if the mask is specified as a data constant with an S, F, or B in the location field (a B designated mask is placed in the field mask pool). As a result, it will not be duplicated within the same group if a library macro or a dependent subroutine uses the same mask or if a mask having the same binary configuration is generated.

All masks which remain constant should be placed in the pool, in case they are used by library routines. However, any mask which is mcdified during the execution of a program should be withheld from the pool.

SECTION XI

ARGUS UPDATING FUNCTION

As stated in Section I, the ARGUS Assembly Program maintains a symbolic program tape (SPT) which contains a file of programs undergoing checkout. Every program on the SPT is in the original assembly language, thus allowing modifications or corrections to be made to the program in this language. The SPT also contains the test data and debugging pseudo instructions (derails) for each program, which may also be modified or corrected during an assembly run. The structure of the SPT is described in Appendix B.

In order to maintain this file, the Assembly Program accepts as input the existing SPT, a deck of cards, and/or one or more reels of tape containing card images. The card or tape input represents new ARGUS programs to be assembled, corrections to existing programs on the SPT, and output (new programs) from the compilers. The outputs from the updating run are a new SPT, requested printed listings and analyses of programs, requested punched cards in ARGUS format, and a list of the programs and segments on the new SPT. These outputs are described in Section XII.

A group of control instructions is used to direct the ARGUS updating process. These control instructions are punched one per card, like machine instructions, but they are not assembled by ARGUS to produce machine words in a program. Each instruction is identified by an operation code of up to eight alphabetic characters which is punched in the command code field.

ARGUS

This instruction must be the first card of every ARGUS input deck. The A and B address fields are not used. The C address field may contain a code describing a standard assembly equipment configuration to be used for all programs which do not specify a particular configuration code (see PROGRAM instruction, page 86). A detailed description of this configuration code is contained in Appendix C. If the C address field is blank, ARGUS assumes the standard configuration to be that of the machine on which the updating run is being made.

Program Directors

A program director card marks the beginning of input for the program whose name is specified in the A address field. The director cards are distinguished by the prefix "U, "

punched in the first two command code columns (columns 11 and 12). The five types of director cards described below are distinguished by the command code following this prefix. The location field, the B and C address fields, and the remarks field are unused except as otherwise noted.

The program name may consist of from one to eight characters. Leading or imbedded spaces are eliminated during input processing. This processing includes checks to prevent duplication of program names on the same tape.

Any input between the ARGUS card and the first program director card is discarded during input processing. All input for each program must follow its program director card, and all cards following a particular program director card are included as part of that program until the next program director card is encountered.

U, ELIMPROG: This card directs ARGUS updating to eliminate the program named in the A address field from the new symbolic program tape. The S/C and line number fields are not used.

U, REASSEMB: This card directs ARGUS to reassemble the program named in the A address field, using the old program on the symbolic tape as a major input to assembly. Any changes which follow this director card are merged with the existing program during updating. The resulting program is reassembled and written on the end of the new symbolic tape, and the old version of the program is eliminated.

If the S/C column is blank, any scientific instruction in the program is replaced by a call for a library routine. Otherwise, scientific operation codes are translated normally.

If the line number field is blank, the line numbers resulting from merging the inputs are preserved in the reassembled output. If any legal characters other than blanks appear in the line number field, ARGUS reassigns line numbers in the reassembled program.

U, CORRECT: This card directs ARGUS to correct without reassembly the program specified in the A address field. S/C and line number field options are the same as on REASSEMB cards.

Correction is a process by which certain classes of simple changes can be made to the old program without requiring the more extensive reassembly process. Correction is

restricted to changes in test data, derails, and certain kinds of one-for-one program word replacements which do not require reallocation of addresses. Wherever possible, the programmer should elect to correct rather than to reassemble his program in order to save computer time. However, if the Assembly Program detects a situation which violates the correction rules, the "CORRECT" instruction is revised and the program is reassembled instead. The situations in which revision will occur are as follows:

1. Occurrence of one or more symbols in the address fields of EQUALS or RESERVE instructions in the program;
2. Reference to more than 100 different symbolic tags in corrections to any segment of the program;
3. Any reference to a tag which was not defined when the program was last assembled;
4. Any reference to a tag which was defined by a TAS instruction in the last previous assembly;
5. Insertion of a new line between existing lines, before the first line or after the last line of any segment;*
6. Deletion (as opposed to one-for-one replacement) of any line;*
7. Addition or elimination of any segment;
8. Replacement of a line by a new line, where the location fields are not identical;*
9. Use of the instructions EQUALS, RESERVE, EVEN, SIMULATE, MODLOC, or MASKGRP, except to replace an identical line;
10. Use of the instructions SETLOC, ASSIGN, TAS, or the LINK constant;
11. Call for a subroutine which was not called in the last previous assembly;
12. Use of mask generation parameters in a field or shift instruction unless the instruction being replaced generated an identical mask;
13. Use of a mask within macrocoding, unless the instruction being replaced used an identical mask;
14. Replacement of any line by another line where the numbers of machine words produced by each line are not identical (e.g., replacement of a macro routine by another macro routine not of the same length, replacement of a FLBIN constant by an EBC, or replacement of a line containing constants by one containing an unequal number of constants);
15. Use of a MASKBASE constant with S,n or F,n in the address fields. Tags must be used even if the constant being corrected used S,n or F,n. Note that a group may be specified by any tag assigned within that group; and
16. Since a printed listing is not provided for programs undergoing correction, the occurrence of any error on an input card will cause the program to be reassembled and listed.

*Exception: Remark cards and words to be loaded into special registers (Z, in location field) may be inserted, deleted, or replaced by other remarks or special register cards even when the location fields are not identical.

S/C and line number field options on CORRECT cards are the same as on REASSEMB cards.

U, NEWVERS: This card directs ARGUS to make a new version of the program whose name is specified in the A address field, giving it the new name specified in the B address field. The new name obviously must not duplicate the name of any other program on the symbolic tape. Except in name, the new program may exactly duplicate the old one, or it may be modified to any desired extent by the input following the NEWVERS card.

Regardless of the degree and nature of change, all new versions are put through the reassembly process. In this respect, NEWVERS has the same effect as REASSEMB except that the old version is not automatically eliminated unless ARGUS is specifically instructed to do so by an ELIMPROG card.

S/C and line number field options on NEWVERS cards are the same as on REASSEMB cards.

U, NEWPROG: This card directs ARGUS to assemble the following input as a new program. The name of the program is specified in the A address field. Aside from the name duplication checks, no reference is made to the contents of the old symbolic tape.

S/C and line number field options are the same for NEWPROG cards as for REASSEMB cards.

Programmer Macro Routine Markers

Two instructions, MACRODEF and FINIS, are used to mark the beginning and end, respectively, of programmer-defined macro routines within programs.

MACRODEF: This instruction must precede each programmer macro routine. The location, address, and line number fields are not used. The S/C column may be used to specify one of the following card check options:

1. If the S/C column contains an "I" (Identification), the contents of columns 74 through 80 of the remarks field are stored (in unjustified form and without space suppression) and used to check all succeeding cards of this programmer macro routine. If the corresponding columns of a succeeding card in this macro routine do not match the stored value, the card is discarded during assembly.
2. If the S/C column contains an "S" (Sort), the contents of the first five line number columns (66-70) of each card in the routine are converted

to a binary number and used as a low-order key. Otherwise, the cards of this macro routine are assigned keys which will preserve the order in which they occur.

3. If both identity checking and sorting are desired, the S/C column contains a "B". If neither is desired, this column is left blank.

The cards following the MACRODEF instruction represent the master macro instruction and the actual macro routine. They are described more fully in Appendix A on "Writing Macro Routines". These cards are subject to the sorting and identification check options specified in the MACRODEF card. If the sorting option was specified, columns 66 through 70 must contain the serial number of each card within the macro routine.

FINIS: This instruction signals the end of a macro routine. An identification check on the contents of the remarks column (74-80) of the FINIS card is made, provided that this option was specified in the MACRODEF card.

Segment Directors

The three types of segment directors defined below all mark the beginning of input for a particular segment of the program. The name of the program to which the segment belongs is written in the A address field and is subject to the same conventions as on the program director cards previously described. The B address field must contain the segment name, which may contain a maximum of seven characters, but which is otherwise handled in the same manner as a program name.

ELIMSEG: This card directs ARGUS to eliminate the specified segment from the new symbolic tape. The names of the program and segment are specified in the A and B address fields, respectively. The S/C and line number fields are not used in this card. No detail cards should follow this director.

SEGMENT: This card marks the beginning of input for the specified segment. The program and segment names are specified in the A and B address fields, respectively. The location and C address fields are not used. ARGUS determines whether this input represents changes to an existing segment or an entire new segment by comparing the program and segment names against a directory of the old symbolic program tape.

The S/C column may be used to specify one of the following handling options:

1. If an "I" is written in the S/C column, the contents of columns 74 through 80 of the remarks field are used to check the identity of all following detail cards up to the next program or segment director, just as in the

MACRODEF instruction.

2. If an "S" is written in the S/C column and the input is a new segment, the contents of all eight line number columns (66-73) on the following detail cards are converted to binary numbers which are used to sort these cards within the segment. If the input represents changes to an existing segment, line numbers must be provided to identify the words to be changed and this input is automatically sorted, regardless of the contents of the S/C column.
3. If a "B" is written in the S/C column, both identity checking and sorting are provided.
4. If the S/C column is blank and the input is a new segment, ARGUS generates line numbers to preserve the existing order of the cards, providing spaces for later insertion of changes as explained on page 16. As stated above, if the input represents changes to an existing segment, it is automatically sorted.

The line number field of the segment instruction may be used to specify certain optional outputs from the assembly process. These outputs, which are described in Section XII, may be produced only when a segment is assembled or reassembled, not when it is corrected. The following codes may be written in any combination, separated by commas. The order in which they are written is immaterial, provided that they are written in the line number field.

Code	Interpretation
S	Produce a new symbolic card deck for this segment from the updated SPT.
L	Produce a symbolic listing of coding for this segment, including all diagnostic printouts.
LS	Produce a symbolic listing for this segment, including diagnostic printouts for all definite errors but suppressing printouts for possible errors.
A[1]	Produce an analyzer of the coding for this segment, including all diagnostic printouts.
AS[1]	Produce an analyzer of the coding for this segment, including diagnostic printouts for all definite errors but suppressing printouts for possible errors.

PROGRAM: This card is a segment director of a special kind, used to introduce the first (or only) segment of a program. It is identical in function to the SEGMENT instruction (above) with the following exceptions:

1. The C address field may be used to specify an assembly equipment configuration code for this program, according to the configuration statement format described in Appendix C. If this field is blank,

[1]Since the analyzer includes a symbolic listing, it is never necessary to specify both.

the standard configuration is used (see the ARGUS card above).

2. If the program logic requires that tape units assigned to a common tape control remain assigned to a common tape control, or if the program contains any reference to a read-write address counter (RAC, DRAC, WAC, or DWAC), the command code (PROGRAM) must be followed by the suffix ", R". This suffix designates the so-called R restriction, which directs the assignment of equipment codes by Executive, as described in the Executive System Manual.

Test Data Directors

Test data belonging to a segment may appear anywhere within the input for that segment. The Program Test System Manual should be consulted for a detailed description of the test data cards which are summarized here.

TESTDATA: This card introduces a set of test data or changes which are to be applied to an existing set. The location field contains the set number, in the range 0 through 7, the A and B address fields contain the program and segment names, respectively, and the C address field contains the first address of the read-in area in high-speed memory, expressed as an absolute decimal address.

ELIMDATA: This card directs ARGUS to eliminate the specified set of test data from the SPT. The location field contains the set number (0-7), and the A and B address fields contain the program and segment names, respectively.

Test Data Detail Cards

1. Distributing Pseudo Instructions. Each distributing instruction contains in its command code field an "X, " followed by the operation code of the instruction. The location field has the format s, r, p where s is the set number (0-7), r is the record number (three decimal digits) and p is the instruction position number (01-20). The cards are subject to the identification check of columns 74 through 80 as specified in the TESTDATA card. A detailed description of the functions and formats of these instructions is found in the Program Test System Manual.

2. Delete Distributing Pseudo Instructions. This card (X, DELETE) directs ARGUS to delete one or more distributing instructions from a set of test data. The location field contains s, r, p (set number, record number, and position number of the first instruction to be deleted). The A address field contains TO. The B address field contains f, h, where f is the record number and h is the position number of the last instruction to be deleted. This card is subject to

the identification check of columns 74 through 80 as specified in the TESTDATA card.

3. Test Data Words. Each test data word card contains in its command code field a D, and the name of an ARGUS data constant or mixed constant containing numeric data in all fields. The card may contain several constants of the same type, as may any constant card. The location field contains s, r, w, (set number, record number and data word position number for the first constant on the card). These cards are subject to the identification check of columns 74 through 80 as specified in the TESTDATA card. Test data words may be altered on a one-to-one basis during an updating run.

4. Delete Data Words. This card (D, DELETE) directs ARGUS to delete one or more data words from an existing set of test data. The location field contains s, r, w, designating set number, record number and data word position number of the first word to be deleted. The A address field contains "TO", and the B address field contains f, h (record number and data word position number of the last data word to be deleted). This card is subject to the identification check.

Debugging (Derail) Pseudo Instructions

A complete description of the formats and functions of these instructions is found in Section III of the Program Test System Manual. The sort and identification check options, specified in the segment director card, may be used with these instructions; if the sort option is desired, columns 66 through 70 must contain the serial number of the instruction within the segment.

ELIMDERL: This card directs ARGUS to eliminate the derail whose serial number is specified in the first five line number columns. If these columns are blank, all derails are eliminated from the corresponding segment. If the segment director card contains an "I" in the S/C field, a segment identification must be punched in columns 74 through 80 of the remarks field. The location and address fields are not used in this card.

Main Coding

Main coding refers to all instructions and constants belonging to a segment. The sort and identification check options, as specified on the segment directors, may be used to check all main coding cards. If the sort option is used, columns 66 through 73 on every card must contain a line number representing the position of that card within the segment.

The line number field must indicate the words to be changed within an existing segment. It may indicate the order of a complete new segment. ARGUS automatically generates line numbers to preserve the original order of new segments if the sort option is not specified.

DELETE: This card directs ARGUS to delete lines of main coding from an existing segment on the symbolic program tape. The line number columns specify the first line to be deleted. If the A and B address fields are blank, only this line is deleted. If, however, the A address field contains the word "TO" and the B address field contains the line number of the last line to be deleted, ARGUS will automatically delete all intervening lines, provided that both specified lines are contained in the same segment. Original five-digit line numbers may be punched anywhere in the B address field; insertion line numbers must contain a decimal point to separate the high-order five digits from the low-order three digits.

ENDARGUS

A card with ENDARGUS punched in the command code field must be used to signal the end of the ARGUS input deck. The contents of all other fields on the card are ignored.

Ordering the ARGUS Input Deck

ARGUS automatically sorts the entire input deck into an order which is convenient for updating, but before the sorting can begin the deck must be in order according to the following rules:

1. All cards belonging to one program must be together, preceded by a program director card.
2. All cards belonging to a generalized programmer macro routine must be together, bounded by MACRODEF and FINIS cards. The master macro instruction must immediately follow the MACRODEF, and the entire routine must precede the first macro instruction referring to it.
3. All cards (test data, information request pseudo instructions, and main coding) belonging to a segment must be together, preceded by a segment director card. In other words, the only cards which may appear between the program and segment directors are macro routines, but macro routines may also appear after a segment director.
4. Test data detail cards must be preceded by a TESTDATA card, and the first card which does not have an "X, " or "D, " in the command code field will signal the end of a group of test data cards. That is, no card other than test data cards may appear within a group of test data cards.
5. If the sort option is not specified (i. e., the cards do not contain line numbers) within a macro routine, the macro routine cards must be in the proper order with respect to one another.
6. If the sort option is not specified for a new segment, the derail pseudo instructions, as well as the main coding, must be in the proper order with respect to one another.
7. Derail pseudo instructions may not appear within programmer macro routines. The appearance of a program director, segment director,

or MACRODEF card within the macro routine will terminate the routine.

ARGUS sorts the individual program decks so that updating for existing programs occurs in the order in which the programs are stored on the SPT, and new programs follow in the order in which they occurred in the input deck. If there is to be a new version of a program, the updating for the new version precedes the updating (if any) for the old version. Within a program, the segments are sorted so that updating for existing segments occurs in the order in which the segments are stored within the program on the SPT, and new segments follow in the order in which they occurred in the input deck within that program. If there is an ELIMPROG or ELIMSEG card for the program or segment, that card will precede any other cards for that program or segment.

Within each program, the program director card is followed by all programmer macro routines belonging to that program (if any). The routines are in the order in which they occurred in the input deck. The order within each routine is:

1. MACRODEF;
2. Master macro instruction;
3. Macro routine coding (If the "S" option is used, the coding is ordered in accordance with the contents of columns 66 through 70, otherwise, it is in its original order.); and
4. FINIS.

The macro routines are followed by the segments of the program. For each segment, the segment director card is followed by any test data for the segment. The test data sets are in order by set numbers. Within each set the order is:

1. ELIMDATA;
2. TESTDATA; and
3. Test data detail words in order by record number. Within each record, the distributing pseudo instructions (in order by instruction position number) are followed by the data words (in order by data word position number). If there is an X, DELETE or D, DELETE card, it precedes any other X or D card with the same position number.

The test data is followed by the derail pseudo instructions. If the sort option was specified, the derail cards are in order according to the contents of columns 66 through 70; otherwise, they are in their original order. If there is an ELIMDERL card in which columns 66 through 70 are blank (eliminate all derails), this card precedes all other derail cards. Any other ELIMDERL card precedes the derail card which has the identical line number.

The main coding is the last element within the segment. If the sort option was specified, these cards are in order according to the contents of columns 66 through 73; otherwise, they are in their original order, except that the END card is always the last card for the segment.

Equipment Requirements for the Updating Run

The normal complement of equipment for the updating run includes a card reader, a printer, and four magnetic tape units. Two tape units are used for the old and the new symbolic program tapes and two are used for work tapes. A card punch may be added if the input to the run includes any requests for punched card output. The input deck may be converted off-line and then read from magnetic tape by ARGUS without the necessity of providing an additional tape unit, unless the amount of input exceeds one tape reel. If either printing or card punching is off-line, a fifth tape unit is required. However, in no case are more than five tape units required by the updating run. Updating uses two banks of high-speed memory and one group of 28 special registers and can be relocated for parallel processing.

SECTION XII

OUTPUT FROM ARGUS ASSEMBLY OPERATION

The primary output of the ARGUS assembly process is a file of programs on magnetic tape in both symbolic assembly language and machine language. Each program in this file includes relocation information and error information. The tape containing the program file is called the symbolic program tape (SPT) and is the communication link between Assembly and the other ARGUS systems programs (as shown in Figure 1, page 2). The SPT is used in the following three ways:

1. It is the input to the next ARGUS updating run;
2. It is the source of information for the program test run (as described in the Program Test System Manual); and
3. It is the source of information for the Executive scheduling run (as described in the Executive System Manual).

In addition to the updated symbolic program tape, the programmer may direct ARGUS to produce any or all of the following types of secondary output. The segment directors (see page 85) are used to specify which, if any, of these secondary outputs shall be produced for each segment assembled:

1. A complete printed listing of the assembled segment, including the line number and the symbolic input form and machine form of each included word, along with any errors detected during assembly;
2. A printed analyzer, including all of the information given in the listing, plus a list of all references to each symbolic tag by line number;
3. A set of punched cards containing the entire segment in assembly language, complete with line numbers and in the correct sequence.

ARGUS Listing

Each page of the ARGUS listing is headed by a line of print which includes the program name, the segment name, and the page number. This is followed by a line which contains the field headers for the various fields of the listing. These two header lines are included in the sample output page shown in Figure 14, page 99. Each line of coding is printed in both the symbolic language of the programmer and the resulting machine language. The line number appears at the left of each line. In general, program words are listed in the format shown in Figure 10. Exceptions are data constants, EQUALS and RESERVE instructions, and remarks cards, which are listed in the formats shown in Figures 11 and 12. Note that if several data constants are combined on a single input card, each such constant is listed as an individual line of print and the line number is repeated as necessary. A derail instruction is followed on the listing by a CAC constant which contains the addresses assigned to the symbolic tags appearing in the derail. The CAC constant does not appear in the assembled program.

If a programming error is detected in a line of coding, the listing for that line includes an asterisk preceding the line number. This line is followed in the listing by an error printout line which indicates the nature of the error(s). Each error is represented in the error printout line by a key which is printed immediately below the field of the symbolic input word in which the error is detected. The various keys which may be printed are described below under "Programming Errors Detected".

The ARGUS listing is followed by a list of the group numbers of all mask groups used by the segment and the mask base addresses assigned to these groups. Finally, the names of all subroutines which are called by the segment are listed, together with the starting address assigned to each subroutine.

Analyzer

The analyzer is an optional index of every reference to every symbolic tag used within a segment. If this option is requested, by means of the segment directors, the listing line for every word which is tagged in the location field is followed by one analyzer line for each program word which references the corresponding tag. The format of an analyzer line, shown in Figure 13, includes the segment number, line number, and operation code (or constant code) of one program word which references the pertinent tag, with the tag itself appearing in the same field in which it appears in the original word. If the reference includes address arithmetic, the address modifier also appears in the analyzer line. Note that the sample output page shown in Figure 14 includes analyzer output.

If the analyzer option is requested, each mask group listed is followed by one analyzer line for each reference to that group. (Here a reference to a mask group is the generation of a mask in that group or the appearance of the group number in a MASKGRP instruction or a MASKBASE constant, since all symbolic mask references have already been listed.) Next, each subroutine listing is followed by an analyzer line for each call instruction to that subroutine. Finally, the analyzer includes an index of all references to peripheral addresses.

Programming Errors Detected

The types of programming and card punching errors which are detected during assembly are listed in Figure 15. For each error detected, an error indication is recorded on the symbolic program tape and printed as part of the ARGUS listing. The Assembly Program detects errors which fall into two broad categories: definite errors and possible errors. The error indications for definite errors are always printed, whereas the programmer may specify on the segment directors that the error indications for possible errors are to be suppressed.

Print Positions	Related Card Cols.	Field	Description
1	-----	Error Indicator	Asterisk indicates the presence of an error.
2-9	66-73	Line Number	May be written by programmer or generated by ARGUS.
10	-----	Separator	SPACE
11-20	1-10	Location	Programmer Notation
21	-----	Separator	SPACE
22-33	11-22	Command Code	Programmer Notation
34	23	S/C	Programmer Notation
35	-----	Separator	SPACE
36-49	24-37	A Address (Symbolic)	
50-63	38-51	B Address (Symbolic)	Programmer Notation
64-77	52-65	C Address (Symbolic)	
78	-----	Separator	SPACE
79	-----	Bank	Bank or group indicator of assigned location.
80-83	-----	Subaddress (Absolute)	Subaddress of assigned location: in decimal if memory location, in absolute ARGUS format (see below) if special register.
84	-----	Separator	SPACE
85-88	-----	Op Address (Absolute)	Absolute value (in decimal) of command code subaddress for shift, simulator, and masked field instructions and mixed constants.
89-96	-----	A Address (Absolute)	Absolute value (in decimal) of address fields, right justified. Complex address assignments shown in absolute ARGUS format. as follows:
97-104	-----	B Address (Absolute)	
105-112	-----	C Address (Absolute)	Direct memory location address 0199 Indexed memory location address 3,026 Special register address N,01,02 Indexed special register address 3,N,14,2
113	-----	Separator	SPACE
114-120	74-80	Remarks	Programmer Notation

Figure 10. ARGUS Listing - General Format

Print Positions	Related Card Cols.	Field	Description
1	-----	Error Indicator	Asterisk indicates the presence of an error.
2-9	66-73	Line Number	May be written by programmer or generated by ARGUS.
10	-----	Separator	SPACE
11-20	1-10	Location	Programmer Notation
21	-----	Separator	SPACE
22-33	11-22	Command Code	Programmer Notation
34-35	23	Separator	SPACES
36-77	24-65	Constant	Programmer Notation
78	-----	Separator	SPACE
79	-----	Bank	Bank or group indicator of assigned location.
80-83	-----	Subaddress (Absolute)	Subaddress of assigned location in decimal.
84-88	-----	Blank	SPACES
89-91	-----	Op Code	OCT (denotes octal notation of machine word)
92-93	-----	Blank	SPACES
94-97	-----	Op Address (Absolute)	Bits 1-12 of machine word (in octal)
98	-----	Blank	SPACE
99-102	-----	A Address (Absolute)	Bits 13-24 of machine word (in octal)
103	-----	Blank	SPACE
104-107	-----	B Address (Absolute)	Bits 25-36 of machine word (in octal)
108	-----	Blank	SPACE
109-112	-----	C Address (Absolute)	Bits 37-48 of machine word (in octal)
113	-----	Blank	SPACE
114-120	74-80	Remarks	Programmer Notation

Figure 11. ARGUS Listing - Data Constants

Print Position	Related Card Cols.	Field	Description
		EQUALS AND RESERVE INSTRUCTIONS	
1	-----	Error Indicator	Asterisk indicates the presence of an error.
2-9	66-73	Line Number	May be written by programmer or generated by ARGUS.
10	-----	Separator	SPACE
11-20	1-10	Location	Programmer Notation
21	-----	Separator	SPACE
22-33	11-22	Command Code	Programmer Notation
34-35	23	Blank	SPACES
36-77	24-65	Expression	Programmer Notation
78	-----	Separator	SPACE
79	-----	Bank	Bank indicator or sign of assigned value.
80-83	-----	Subaddress (Absolute)	Subaddress or integers of assigned value in decimal.
84-88	-----	Blank	SPACES
89-96	-----	Complex Address (Absolute)	Complex assignment in absolute ARGUS format (see Figure 10) if tag also has a complex assignment.
97-113	-----	Blank	SPACES
114-120	74-80	Remarks	Programmer Notation
		REMARKS CARDS	
2-9	66-73	Line Number	Written by programmer or generated by ARGUS.
25-104	1-80	Remarks	Card image of remarks card.

Figure 12. ARGUS Listing - Equals and Reserve Instructions and Remarks Cards

Print Positions	Field	Description
1-24	Blank	SPACES
25-34	Command Code	Operation code (or constant code) of program word in which the tag appears. Also the tag with or without address modifier if the tag appears in the command code field of the program word.
36-49	A Address	Tag with or without address modifier if the tag appears in the A address field of the program word.
50-63	B Address	Tag with or without address modifier if the tag appears in the B address field of the program word.
64-77	C Address	Tag with or without address modifier if the tag appears in the C address field of the program word.
78-79	Segment Number	The segment number of the segment in which the reference appears.
80	Blank	SPACE
81-88	Line Number	May be written by the programmer or generated by ARGUS.
89-120	Blank	SPACES

Figure 13. ARGUS Listing - Analyzer Lines

```
PROGNAME SEGNAME                                                                                    PAGE nnn
E LINE NO LOCATION   OPERATION    S A ADDRESS      B ADDRESS      C ADDRESS      B LOC MASK     A      B      C      REMARKS

 00065000 COMPUTE    WA,MASK1     C ASSIGN         ADDARITH       0,4            10500 0128     0502   0503   0,004
                        TS                                       COMPUTE        00 00006000
                        WD           COMPUTE+3                                  00 00138000
                        SS                                       COMPUTE+4      COMPUTE+3      00 00279000
*00066000               TX,MASK2     SYMBOL           -          LISTIMAG+5     10501 0129     N,09,01  ----   3,006
                                    *USAGE                                  *AA
 00067000 STORETAG      ALF          TAG                                      10502     OCT  6321 2715 1515 1515
 00068000 B,MASK1       DEC          000GGG000GGG                             10960     OCT  0000 7777 0000 7777
                           WA,MASK1                                           00 00065000
 00069000 ASSIGN        DEC          GGGGGG000801                             10503     OCT  7777 7777 0000 4001
                           WA        ASSIGN                                   00 00065000
 00070000 ADDARITH      OCT          777500020016774 5                        10504     OCT  7775 0002 0016 7745
                           WA                        ADDARITH                00 00065000
 00071000 SYMBOL        ASSIGN       N,X1,1                                                 N,09,01
                           TX        SYMBOL                                   00 00066000
 00072000 LISTIMAG      ASSIGN       3,1                                                       3,001
                           TX                                    LISTIMAG+5  00 00066000
 00073000 A             EQUALS       ABCDE+1234/C+BCDEFGH-HABC                              +0256
 00074000 CAC1          CAC        + OUTBUFA        OUTBUFB      LSTA           10505          10801  11001  11201
 00075000 Z,X1          SPEC                                     INPUTBUF       10506                        11401
 00076000 AB            RESERVE      A                                        10507          +0256
 00077000 ABC           EQUALS       ADDARITH                                 10504
 00078000 ABC           ASSIGN       Z,X7,31                                                      Z,15,31
 00079000 COMPUTE       ASSIGN       N,X0,16                                  10500                N,08,16
 00080000                  R, AND THIS IS JUST A SAMPLE
```

NOTES:

Line 65 - is an example of the general printed format for an instruction, showing the symbolic coding written by the programmer and the absolute location and address fields of the assembled instruction. The three analyzer lines which follow show references to the tag appearing in the location field.

Line 66 - contains two errors, reference to a special register address in a masked instruction, and use of address arithmetic with a tag assigned a complex address. The error line which follows provides the programmer with diagnostic information on the errors.

Lines 67-70 - illustrate the octal format of the absolute machine word interpretation.

Lines 71-79 - illustrate various examples of control constants and control instructions.

Line 80 - is a remarks card as printed on the listing.

Figure 14. Sample ARGUS Listing (With Analyzer)

Printout	Field	Error Condition
*AA	Command, A, B, or C	Illegal use of address arithmetic (e.g., with a tag assigned to a special register address).
*ASSIGN	Command	Address or augmenter in simulator instruction does not end in 111.
*ASSIGN	A	Tag used in A address field of SETLOC has improper assignment.
*ASSIGN	Command, A, B, or C	Use of a tag inconsistent with its assignment (e.g., symbolic augmenter not equated to an integer, tag in a masked instruction assigned to a special register, etc.).
*BNK ERR	A, B, or C	Result of address arithmetic is negative or greater than the largest address in the system.
*BNK ERR[1]	A, B, or C	Tag in an address field of an instruction is assigned to a different bank than the instruction.
*BLANK	Location	Location field is blank in EQUALS, ASSIGN, or TAS.
*CONFIG	Location	Illegal tag (illegal characters or all numeric).
*CONFIG	Command	Generated mask parameters exceed limits or resultant mask goes beyond right end of word.
*CONFIG	Command	A mask is designated incorrectly.
*CONFIG	Command	For SPEC, CAC, CONTROL, SETLOC, MASKGRP, or END, the characters following the command code are neither blank nor one of the allowed configurations.
*CONFIG	A	Illegal characters within a data constant.
*CONFIG	A	Parameters in a macro instruction do not correspond to those in the master macro instruction or they cross field boundaries in the coding.
*CONFIG	A, B, or C	Z, or N, not followed by a valid special register name or number.
*CONFIG	A, B, or C	Non-numeric characters following C, or X, .
*CONFIG	A, B, or C	Non-numeric characters in address modifier or increment.
*CONFIG	A, B, or C	Non-numeric characters in number of shifts following A, B, or D.
*DEVICE	Command	Peripheral address inconsistent with instruction (e.g., WF to a card reader, etc.).
*DEVICE	Command or A	Peripheral address in peripheral instruction or TAC not available according to the equipment configuration.
*DUPLCAT	Location	A tag appears in the location field more than once with conflicting assignments (note that tags may appear more than once without conflicting assignments).
*GRP OER	Command	New mask generated when current group is full.

Figure 15. Programming Errors Detected During Assembly

Printout	Field	Error Condition
*ILLEGAL	Location	Any character other than Z, F, S, B, X, L, R, or P followed by a comma in the location field.
*ILLEGAL	Location	Z, not followed by a legal special register name.
*ILLEGAL	Location	F, S, B, or L not followed by a tag.
*ILLEGAL	Command	Illegal operation code.
*ILLEGAL	Command	Call for a macro routine which is not in the library.
*MEMOFLO	Location	A segment overflows the memory of the computer.
*MSK ERR[1]	Command	Use of a shift mask in a field instruction or vice versa.
*MSK ERR[1]	Command	Symbolic reference to a mask not in the current group.
*MSK ERR	C	A dependent subroutine is called for twice with two different mask groups in control.
*OVERFLO	Location or Command	More than eight characters in a tag.
*OVERFLO	Location	New mask designated when current group is full.
*OVERFLO	A	Data constant larger than the maximum value allowed.
*OVERFLO	A, B, or C	Address modifier greater than 2047 or 16, 383; augmenter greater than 255 (254 for X7); increment greater than 31; number of shifts greater than 8, 11, 12, 44, 48, or 63 (as allowed); number of words transferred greater than 63, etc.
*OVERLAP	Location	Overlap of subsegments when programmer has specified the initial locations.
*REJECTD	Location	Illegal tag combination in EQUALS or RESERVE (e.g., result has dimension >1 in EQUALS or > 0 in RESERVE, tag was not defined previously, etc.).
*REJECTD	Location	A subsegment which was previously normal is now specified as common.
*UNASYND	Command, A, B, or C	A tag which does not appear in the location field is referenced in an address field.
*UNASYND	A	Symbolic augmenter in ASSIGN or TAS or tag in A address field of SETLOC not defined previously.
*USAGE	Location	Location field is not blank in MASKGRP or END.
*USAGE	Location	Location field is Z, X, F, S, or B in SETLOC, MODLOC, EVEN, or SIMULATE.
*USAGE	Location	Location field is Z, F, S, or B in RESERVE.
*USAGE	Command	A mask is designated in an instruction which cannot be masked.
*USAGE	Command	Ambiguous field type in a generated mask ("A" or "D" not specified).
*USAGE	Command	Peripheral operation code not followed by a valid configuration.

Figure 15. Programming Errors Detected During Assembly (cont)

Printout	Field	Error Condition
*USAGE	Command	Print operation code not followed by blank, M, or MR.
*USAGE	Command	Multiprogram control op code not followed by blank or H.
*USAGE	Command	For LINK, MASKBASE, TAC, SEGNAME, MODLOC, EQUALS, RESERVE, ASSIGN, TAS, EVEN, or SIMULATE, the command code is not followed by blanks.
*USAGE	A	A address field of EQUALS, RESERVE, CONTROL, TAC, or MODLOC is blank.
*USAGE	A, B, or C	Illegal address type for instruction (e.g., special register address in masked instruction, indexed address in SPEC, etc.).
*USAGE	A, B, or C	Address field which should be blank is not blank (e.g., A or B address field of SPEC, C address field of MASKGRP, etc.).
*	S/C	Contents of S/C column not legal for instruction.
[1]Possible Errors		

Figure 15. Programming Errors Detected During Assembly (cont)

SECTION XIII

LIBRARY ROUTINES

The ARGUS system includes a library of checked-out macro routines and subroutines representing frequently used coding which is preserved for easy insertion into new programs. Each macro routine or subroutine in the library is requested by means of a pseudo instruction which specifies the desired routine and all parameters required for its execution. These ARGUS pseudo instructions may be included in a program as easily as machine instructions. When a program is assembled, ARGUS recognizes each pseudo instruction, obtains the corresponding coding from the library, and incorporates it into the program. The specifications of every routine in the library are documented on a library routine specification sheet prepared by the programmer who wrote the routine.

Macro Routines

Macro routines are stored in the ARGUS library in the symbolic language in which they are originally written and in generalized form. When a program being assembled requests a macro routine from the library, the routine is assembled into machine language and specialized to meet the needs of the requesting program, according to the parameter values in the pseudo instruction. The routine may be designed to be inserted either entirely in sequence at the point where the pseudo instruction was written or partly in sequence and partly out of sequence. At least one word of the routine must be inserted in sequence at the point of the pseudo instruction in order to transfer control to the out-of-sequence portion. If the routine contains more than one in-sequence word, the programmer must exercise care in using address arithmetic in the vicinity of the pseudo instruction. If the routine contains any out-of-sequence coding, this coding is stored at the end of the subsegment which includes the requesting pseudo instruction. The programmer must see that any subsegment which contains such out-of-sequence coding can be stored by Assembly within a single memory bank.

The pseudo instruction which is used to call a macro routine is called a macro instruction and has the following format. The use of a tag in the location field is optional. If the

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

1 LOCATION 10	11 COMMAND CODE 22	S/C 24	A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	66 REMARKS LINE NUMBER 73 74 80
tag	L, NAME, p1/p2	S/C	p3/p4	. . . . .	. . . . /pn	

macro instruction is tagged, ARGUS assigns this tag to the first word of the assembled macro

routine in the object program. Any reference to this tag within the same program segment refers to the first word of the assembled routine. The letter "L" in the command code field is a control character which designates that the following code is the name of a routine in the library. The control character is followed by a comma and the name of the desired routine. The name may contain up to eight alphanumeric characters, of which at least one must be non-numeric. If one or more parameters are specified in the command code field, the name must be followed by a comma.

The S/C column normally designates the same sequencing counter that selected the macro routine. This causes the routine to be executed under control of the specified counter, except that any sequencing counter designation(s) within the routine will override the designation in the macro instruction. However, if the routine includes coding to set up the sequencing counter designated in the macro instruction, which can be determined from the specification sheet, then the programmer is free to specify either counter in the macro instruction. Note that only a macro routine which contains such coding can be entered from different sequencing counters within the same program.

The codes $p_1/p_2/$ through p_n represent the values of the various parameters required by the macro routine. The type and format of each parameter required by a given routine may be obtained from the specification sheet. A macro instruction may contain up to 25 parameters, but the number of characters which can be written in one field is limited to 12 in the command code field and 14 in each address field. Individual parameters must not cross field boundaries and must be separated from each other by slashes or by the end of a field.

ARGUS specializes a macro routine when it is assembled into an object program by inserting the parameter values stated in the macro instruction. For example, p_1 might specify the number of words in each item of a table, in which case it could be used in the macrocoding as an increment to a special register. Another parameter might be the location of the beginning of the table. (See Appendix A for other examples of parameters.) The formats of parameter values are not necessarily the same for each appearance of the macro instruction. For example, if one of the parameters in the routine is used to reference a mask, the parameter value might be a symbolic tag in one instruction and mask generation information in another. However, when the parameter represents a quantity such as an increment to a special register, for which there is only one acceptable format, the parameter value must have that format in every macro instruction which requests the corresponding routine.

Programmer-Defined Macro Routines

In addition to the use of library macro routines, the programmer may code a macro routine to perform some function which is required several times in his program. Such a routine, called a programmer-defined macro routine, is included once in the card deck for the program in which it is used. Although a programmer macro routine may be called any number of times by the program with which it is assembled, it is not available for execution by any other program (unless it is added to the library as described in Appendix A).

The coding of a programmer macro routine is identified by preceding and following it with the instructions MACRODEF and FINIS, respectively (as described on page 84). The MACRODEF instruction must be followed immediately by a master macro instruction containing a control character of "P" (to designate that the routine is a programmer macro), followed by the name of the routine and the tags of all required parameters. The master macro instruction defines the format of all requesting macro instructions for that routine. The macro-coding is written in the same manner as a library macro routine (see Appendix A).

A programmer macro routine may contain up to 2048 cards. It must be included in the program deck prior to the first macro instruction for that routine. A macro routine is available to any segment of the program after the appearance of the routine in the input deck. The macro instruction which calls for the routine follows the format laid down by the master macro instruction written with the routine (including a control character of "P"), and also supplies the values of all required parameters. If a deck of changes to an existing program includes any reference to a programmer macro routine, this routine must precede the reference in the input deck. This is because ARGUS derives all change information from the input deck and does not refer to the old symbolic program tape to obtain the referenced routine.

Subroutines

The ARGUS library of routines includes subroutines as well as macro routines. A subroutine differs from a macro routine in several important respects.

1. It is assembled into machine language before it is added to the library;
2. It is inserted only once into each program segment in which it is executed, regardless of the number of times that it is executed within that segment;
3. It is inserted out of sequence from the main coding and reached by means of a transfer of control. A second transfer returns control to the main coding when the subroutine is completed; and
4. It is specialized according to the stated parameter values when it is executed, rather than when it is inserted into the program.

Since a subroutine is stored out of sequence from the main program, it requires a linkage, or calling sequence, to make the parameter values available to the subroutine, transfer control to the routine, and then transfer control back to the main coding after the subroutine is completed. The ARGUS system makes use of macro routines as calling sequences for subroutines. Every subroutine in the library uses one or more library macro routines as its calling sequence. Like any other macro routine, the calling sequence is inserted into the program every time that the subroutine is requested. It may be inserted entirely in sequence or it may consist of an in-sequence portion and an out-of-sequence portion. The format and location of the calling sequence are decided by the programmer who writes the subroutine. The flexibility of subroutines is increased by the ability to design different macro routines for calling sequences.

The pseudo instruction which is written to execute a subroutine is known as a call instruction. Since this instruction calls a macro routine which sets up the desired subroutine, it has the same format as any other macro instruction. The control character is always an "L", however, since all subroutines are library routines. A subroutine may be performed several times within the same program segment, using different parameters each time. However this cannot be done by a programmed modification of the call instruction. A separate call instruction must be written for each set of parameters to be used.

As described in Appendix A, a subroutine may be designed to be entered from either sequencing counter and to return control to either counter after execution. Sequencing counter control may be changed within the subroutine, provided that the previous contents of the counters are preserved. In addition, a subroutine may be designed to operate with variables which are stored in a bank(s) different from that in which the subroutine is stored and also different from the bank in which the call instruction is stored. The locations of such variables are specified as parameters. Finally, a subroutine may be designed to be either independent of or dependent upon the programmer's mask groups. Independent subroutines set up their own mask groups and restore the programmer's mask groups after they are executed, while dependent subroutines use the programmer's mask groups. The type of subroutine, dependent or independent, and the number of masks required are indicated on the specification sheet. The same mask group must be in control every time a given dependent subroutine is called within a program segment.

APPENDIX A

WRITING LIBRARY ROUTINES AND THE USE OF LAMP

Writing Macro Routines

A macro routine is written in generalized form, using symbolic tags to represent all parameters of the routine which may vary from one execution to another. The values of these parameters to be used in a given execution are included in the macro instruction which calls the routine. Each macro routine is preceded by a master macro instruction which defines the format of all macro instructions for that routine. Each parameter tag used in the routine must appear in the master macro instruction. When the routine is called, the value of each tag appears in the corresponding position in the requesting pseudo instruction. When the routine is assembled into the object program, it is automatically specialized by replacing each parameter tag within the routine by the corresponding parameter value. As mentioned in Section XI, a macro routine must be preceded by a MACRODEF control card and followed by a FINIS control card when it appears in the ARGUS input deck. The macrocoding itself may consist of from 1 to 2048 cards.

A parameter may be a constant, a field of an ARGUS language instruction, or any portion of a field down to a single character or digit. A phrase of an ARGUS instruction is defined as any part of the instruction which is always bounded by punctuation characters (period, comma, plus, minus, asterisk, or slash) or by the beginning or end of a field. For example, special register designators, augmenters, increments, address modifiers, and operation codes are all phrases. Some fields (e.g., a symbolic tag without modifier) contain a single phrase, while others (e.g., a complex address) contain several phrases. It follows, therefore, that a parameter may represent a phrase, a portion of a phrase, or a group of consecutive phrases. The type and format of the information which the parameter represents is noted on the library routine specification sheet by the programmer who writes the routine.

When a parameter represents only a portion of a phrase, it is necessary to designate the boundary between the parameter and the balance of the phrase. The special symbol (5) (punched as an 8, 5 combination) performs this function within the macrocoding. For example, a macro routine contains the following decimal constant consisting of a plus sign, four zeros, and a seven-digit parameter tagged PAR1.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

LOCATION	COMMAND CODE	S/C	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS / LINE NUMBER
	DEC		+0000 ⑤ PAR1			

The specification sheet for this routine states that PAR1 represents an integer of seven decimal digits. The macro instruction which executes this routine contains a parameter value in the proper format. When the routine is specialized, this value replaces the parameter tag to form the complete constant.

A parameter tag may not include any of the punctuation characters which serve as phrase boundaries (see above) or the character ⑤. A parameter value may not contain a slash or a ⑤, but it may contain any other valid punctuation character. When a line of macrocoding is specialized, phrase boundaries (except ⑤) are retained and parameter tags are literally replaced by their assigned values, including any phrase boundaries which those values may include.

In most cases, each field of a macrocoding word is specialized separately. This limits the number of characters in a parameter value, since the capacity of a field is fixed at 12 characters for the command code and 14 for each address field. Note, however, that data is permitted to cross field boundaries in the control instructions EQUALS and RESERVE, in the TAC constant, and in all of the data constants. Since a parameter value may not exceed one field in length, no phrase which is too long to store in one field may be represented by a single parameter. For example, if a 16-character octal constant appears as a parameter in a macro routine, such a constant must be represented by two parameter tags, since it is too long to store in a single field. The first word in the following example is a master macro instruction in which TAG1 and TAG2 represent two portions of an octal constant shown in the second word. When the programmer writes a macro instruction of the form of the third word, this constant is specialized according to the stated values and appears as shown in the fourth word.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION	COMMAND CODE	S/C	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS / LINE NUMBER
1		L, CONVERT		TAG1	TAG 2		
2		OCT		TAG1 ⑤ TAG2			
3		L, CONVERT		77777773	33333777		
4		OCT		77777773333337	77		

Alphanumeric constants within macrocoding are limited to left-justified series of characters with no intervening space characters. An alphanumeric constant may contain a parameter provided that neither the parameter tag nor its assigned value contain any spaces.

The location field of the first macrocoding word should be left blank, as any tag written in this field is automatically replaced by the location field contents from the macro instruction, even if those contents are all blanks. The location field should also be left blank in all succeeding lines of macrocoding. However, if it is necessary to tag a macrocoding word, a parameter tag should be used so that it can be varied for each execution of the routine. Otherwise, an illegal duplication of tags will result if the macro instruction is written twice within the same segment. Macrocoding words are referenced by using address arithmetic with letters "C" and "X", as described on page 23.

The S/C column in a macrocoding word may contain an "S", a "C", or a blank to specify that the next instruction is to be executed under control of the sequence counter, the cosequence counter, or the sequencing counter designated by the programmer in the macro instruction. However, the controlling counter must be set to the proper value before any instruction can be executed from it. A macro routine may be written so that it must always be executed under control of the same counter that selects it, thereby using the setting already in effect. In this case, the specification sheet must inform the user always to write the macro instruction so that it designates the same counter by which it was selected. A macro routine may be written to set the controlling counter explicitly, so that it may be entered from either counter. In this case, if the routine is to return control to the counter specified in the macro instruction, then at least the last instruction to be executed must contain a blank in the S/C column. A special case is a macro routine which exits under control of the counter designated in the macro instruction to the address which was stored in this counter when the routine was entered. Such a routine must first preserve the contents of one counter, then operate out of sequence under control of the preserved counter, and finally restore the preserved counter and give control to the counter specified in the macro instruction.

Any masks which are required by a macro routine may be either generated or designated within the routine. A mask may be generated by specifying the customary three items of information M_1, M_2, and M_3 (see page 40). The programmer may state the values of these three phrases in the macrocoding or he may represent any or all of them by parameters. A mask may be designated by writing an "F" followed by a comma and a number from 0 to 31 (for a field mask) or an "S" followed by a comma and a number from 0 to 63 (for a shift mask) in the location field. Type "B" masks for use with both field and shift instructions are not permitted

within macrocoding. A designated mask is referenced in an instruction by using the mask number exactly as though it were a mask tag. It is not necessary to assign mask numbers in the order in which the masks are designated. Contrary to normal usage, however, the line which designates a macrocoding mask must follow the last reference to the mask. Note that all masks which are used by a macro routine are included in the mask groups which are in control when the routine is executed. To facilitate the accommodation of macrocoding masks within these groups, ARGUS maintains a mask pool and eliminates any duplication among macrocoding masks, masks used by dependent subroutines, and all generated masks, whether generated in the main coding or in a library routine.

If a macro routine is written to be added to the library of routines, the master macro instruction should contain a control character of "L" in the command code field, and the routine should be included in the deck of cards to be processed by LAMP (the Library Additions and Maintenance Program). After LAMP has added the routine to the library, it can be referenced from any program being assembled. If the routine is written for use as a programmer macro, the master macro instruction should contain a control character of "P", and the routine should be included in the object program deck prior to the first macro instruction by which it is called. In this case, the routine is not available for reference from other programs. However, a programmer macro routine may also be included in the LAMP input deck and added to the library without the necessity of altering the control character.

EXAMPLE: Figure A-1 shows the coding of a macro routine called SRCHEQU as it is written in generalized form. Lines 1 and 12 contain the control instructions MACRODEF and FINIS, respectively. Line 2 is the master macro instruction containing all parameter tags used in the routine. The type and format of each parameter are listed separately on the library routine specification sheet (shown in Figure A-2). Figure A-3 shows a typical macro instruction which might be written to execute the routine SRCHEQU, followed by the routine in specialized form, as it would be inserted into the object program. Note that each parameter tag is replaced by the corresponding value as given in the macro instruction.

Writing Subroutines

A subroutine is written in the form of a program segment, assembled, and placed on the symbolic program tape. After the routine is checked out, it is added to the library where it is available for inclusion in any program being assembled. A subroutine is called dependent if it uses the mask groups currently in control in the object program when it is performed. A subroutine is called independent if it either (1) sets up and uses its own mask groups, preserving and restoring the contents of the mask index register, or (2) does not require any

Honeywell

Electronic Data Processing

ARGUS CODING FORM

PROBLEM MACRO EXAMPLE PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	REMARKS 66 LINE NUMBER 73 74 80
1		MACRODEF					CONTROL CARD
2		L,SRCHEQU,M		ARG/SR	TABLE/SIZE	NOMATCH	MASTER MACRO INSTRUCTION
3		TX		X,+0		Z,SR	SET BEG. OF TABLE
4		TX		X,+1		Z,AU1	END OF TABLE → AU1
5		TX		ARG		N,AU1	ARG → END OF TABLE
6		NA		N,SR,M	ARG	C,+0	ITEM = ARG? → NO ↓ YES
7		WD		Z,SR	X,+2	Z,SR	SET ADDR. OF ITEM
8		LA		X,+1	Z,SR	NOMATCH	END OF TABLE → YES ↓ NO
9	X,	SPEC				TABLE	CONTINUE IN SEQUENCE
10	X,	SPEC				TABLE + SIZE	
11	X,	FXBIN		-M			
12		FINIS					CONTROL CARD
13							
14							
15							
16							
17							
18							
19							
20							

Figure A-1. Sample Macro Routine in Generalized Form

INDEX:
SEARCH CODE: SRCHEQU
DATE: 7/11/60
PAGE: 1

LIBRARY ROUTINE SPECIFICATION SHEET

ROUTINE NAME:	SRCHEQU
BRIEF DESCRIPTION:	This macro routine will search a table for equality on a particular word, continue in sequence if the word is found, or jump to a specified location if not. The location in which the word was found will be left in a special register specified by the programmer. The routine will compare on every nth word of the table, where $1 \leq n \leq 31$. One location must be reserved at the end of the table for use by the macro routine.
PROGRAMMER:	
PROGRAM TYPE:	MACRO X SUBROUTINE
MASK GROUP DEPENDENCE:	DEPENDENT INDEPENDENT
NUMBER OF MASKS USED (IF DEPENDENT):	SHIFT 0 FIELD 0
HSM LOCATIONS USED:	9
SPECIAL REGISTERS USED:	One specified by parameter, and AU1.
SPECIAL REGISTERS RESTORED:	YES NO X
TIMING:	108 + 24x microseconds if word is found, or 108 + 24y microseconds if not, where x = number of items preceding the desired item +1, and y = the total number of items in the table +1.
PERIPHERAL DEVICES:	None
ERROR INDICATIONS AND ACTIONS:	None
OTHER LIBRARY ROUTINES USED:	None

Figure A-2. Specification Sheet for Macro Routine SRCHEQU

INDEX:
SEARCH CODE: SRCHEQU
DATE: 7/11/60
PAGE: 2

LIBRARY ROUTINE SPECIFICATION SHEET (cont)

PSEUDO INSTRUCTION FORMAT:

LOCATION	COMMAND CODE	S/C	A ADDRESS	B ADDRESS	C ADDRESS
	L, SRCHEQU, M		ARG/SR	TABLE/SIZE	NOMATCH

PARAMETER DESCRIPTION:

Symbol	Type	Description
M	Literal	Decimal number (unsigned) - The number of words in each item of the table, ≤ 31. The routine will compare on every Mth word.
ARG	Symbolic, Indirect, or Indexed Address	Location of word to be searched for.
SR	Name (or number) of a Special Register	This special register is used in searching the table and will contain the location of the word searched for, when it is found.
TABLE	Symbolic Tag - Direct Memory Location	Location of the beginning of the table.
SIZE	Literal	Number of words in the table. Must be a multiple of M.
NOMATCH	Symbolic, Indirect, or Indexed Address	Location to jump to when word is not found.

Figure A-2. Specification Sheet for Macro Routine SRCHEQU (cont)

Honeywell
Electronic Data Processing

ARGUS CODING FORM

PROBLEM MACRO EXAMPLE - Continued PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	66 REMARKS 73	74 80
1		L,SRCHEQU,3	C	PRODUCT/R2	PRICETBL/51	WRONGDPT	SAMPLE MACRO INSTR	
2		TX	C	X,+0		Z,R2		
3		TX	C	X,+1		Z,ALL		
4		TX	C	PRODUCT		N,ALL		
5		NA	C	N,R2,3	PRODUCT	C,+0	MACRO CODING AS	
6		WD	C	Z,R2	X,+2	Z,R2	INSERTED INTO	
7		LA	C	X,+1	Z,R2	WRONGDPT	PROGRAM, REPLACING	
8	X,	SPEC				PRICETBL	MACRO INSTRUCTION	
9	X,	SPEC				PRICETBL+51		
10	X,	FXBIN		-3				
11								
12								
13								
14								
15								
16								
17								
18								
19								
20								

Figure A-3. Macro Instruction for Sample Routine and Resulting Specialized Coding

masks. Any segment or program on the SPT may be designated as a subroutine and added to the library, provided that it conforms to the following language restrictions:

1. No special register tags appear in the location field;
2. No link tags are used;
3. No reference is made to a special register group indicator;
4. All peripheral codes are expressed as parameters so that they can be varied for each individual execution; and
5. If the subroutine is to be dependent, it does not contain any type "B" masks and no mask is referenced as an instruction operand.

The assembled subroutine may or may not be a part of a program containing other segments. Although a subroutine must be added to the library as a single program segment, it may be convenient for checkout purposes to write the subroutine as several segments, directing LAMP to combine these segments into one. LAMP combines segments to form a subroutine by a simple overlay process; therefore, the programmer must insure that all segments of a subroutine are capable of being in memory simultaneously and without conflict.

Subroutines may be nested to any desired level. In other words, a subroutine may contain the call instruction of a second subroutine, which may in turn contain another call instruction, etc. Any combination of dependent and independent subroutines may be nested. A subroutine may also contain a macro instruction, although a macro routine is not permitted to include a library pseudo instruction.

As mentioned in Section XIII, every subroutine is stored out of sequence and reached from the main coding by means of a calling sequence. ARGUS uses macro routines as calling sequences. Therefore, to execute a subroutine the programmer writes the macro instruction of the desired calling sequence. The calling sequence is inserted and specialized at assembly time; the subroutine itself is inserted out of sequence, in generalized form, and only once in each program segment in which it is to be executed. When the calling sequence is executed, it specializes the subroutine according to the parameter values stated in the call instruction and then gives control to the subroutine. When the subroutine is completed, control is returned to the main coding via the calling sequence.

Every subroutine calling sequence contains a subroutine call constant, written with SUBCALL in the command code field, blank A and B address fields, and the name of the desired subroutine in the C address field. This constant, which directs ARGUS to insert the routine named, is replaced at assembly time by a special address constant containing the entry address of the subroutine. If the subroutine is to be entered at any point other than the beginning, the subroutine name is modified by address arithmetic in the SUBCALL

constant. If two or more entry points may be used for different executions of the subroutine, the desired entry must be designated by means of a parameter.

For each subroutine which is added to the library, a calling sequence must be provided. ARGUS can be directed to generate either of two standard calling sequences, known as type 1 and type 2, respectively, which fulfill the requirements of many common subroutines. If the parameter requirements of a subroutine cannot be met by either of these standard calling sequences, the programmer must code a special calling sequence to be added to the library along with his subroutine. When a subroutine is added to the library, LAMP either generates the requested calling sequence or processes the special sequence provided; in either case, the calling sequence is added to the macro routine area of the library. Every subroutine consists of three sections: the entry, the body, and the exit. The body of the subroutine is the coding which performs the function for which the routine is written. The entry and exit form linkages from the calling sequence to the subroutine and from the subroutine back to the calling sequence to return to the main coding. The programmer who uses a standard (generated) calling sequence must be familiar with the generated coding in order to prepare his entry and exit.

Type 1 Calling Sequence

The type 1 calling sequence may be used with a subroutine requiring two or three parameters, each of which is a single-word variable and none of which are literals. The input parameters (argument locations) are specified in the A or B address field or both, the output parameter (result location) in the C address field. Only one parameter may be specified in each address field and none may be specified in the command code field.

The coding of the type 1 calling sequence is given in Figure A-4. The parameter values from the call instruction are represented by the quantities in brackets. This calling sequence performs the following functions:

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	REMARKS 66 LINE NUMBER 73 74 80
1		TS	C	Z, CSC	X, +6	X, +0	
2	X,	TX	C	X, +5		Z, AU1	
3	X,	EX	C	-	[B ADDR. PARAM.]	-	
4	X,	TS	C	[A ADDR. PARAM.]	-	N, AU1	
5	X,	TS	C	-	[C ADDR. PARAM.]	-	
6	X,	TX	S	X, +2		Z, CSC	
7	X,	SUBCALL				SUBR1	
8	X,	RESERVE		1			

Figure A-4. Type 1 Calling Sequence

Word 1 preserves (CSC) and resets CSC to X, +0, the address of the first out-of-sequence word in the calling sequence. Word 1 is located in sequence and replaces the call instruction.

Word 2 stores the subroutine entry address in AU1.

Word 3 stores the B address parameter in the mask register, by means of inactive addressing.

Word 4 stores the A address parameter in the low-order product register, by means of inactive addressing. It then transfers the subroutine entry address from AU1 to the CSC and gives control to this counter to enter the subroutine.

Word 5 is the re-entry to the calling sequence from the subroutine. It transfers the result from the low-order product register, where it was stored by the subroutine, to the location specified by the C address parameter.

Word 6 restores the CSC to its preserved setting and returns control to the main coding. The next instruction is selected by the sequencing counter specified in the S/C column of the call instruction.

Word 7, the SUBCALL constant, is in the form of a special address constant and makes the subroutine entry address accessible to the calling sequence.

Word 8 reserves the temporary storage location for (CSC).

Each subroutine which utilizes a type 1 calling sequence must include an entry which obtains the parameter values and places them in the subroutine, and an exit which makes the result available and returns control to the calling sequence.

ENTRY: The entry must include at least the following three words and a constant of all binary ones. If the subroutine is independent and requires masks, the entry must also preserve (MXR) and set up the required mask group(s).

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION 1–10	COMMAND CODE 11–22	S/C 23	A ADDRESS 24–37	B ADDRESS 38–51	C ADDRESS 52–65	REMARKS 66–80 LINE NUMBER 73–80
1		TS	C	–	ARGA	–	
2		TS	C	Z,CSH	CSHSAVE	–	
3		EX	C	ALLONES	–	ARGB	

Word 1 transfers the A address parameter from the low-order product register, where it was stored by the calling sequence, into a temporary location such as ARGA.

Word 2 preserves (CSH) for exit purposes.

Word 3 transfers the B address parameter from the mask register into a temporary location such as ARGB. This word may be omitted if there is no B address parameter.

BODY: The body of the subroutine uses the input parameters stored in ARGA and ARGB to perform the subroutine function, and stores the result in a temporary location such as RESULT, where it is available to the exit.

EXIT: The exit must include at least the following two words. If the subroutine sets up its own mask group(s), the exit must also restore the previous contents of the MXR and set up the main program mask groups previously in control.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION	COMMAND CODE	S/C	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1		TX	C	CSHSAVE		Z, AU1	
2		TS	C	RESULT		N, AU1	

Word 1 transfers the previously saved contents of the CSH into AU1.

Word 2 transfers the result into the low-order product register, then transfers the previous contents of the CSH from AU1 to the CSC and gives control to the CSC to return to the calling sequence.

Type 2 Calling Sequence

The type 2 calling sequence may be used with a subroutine requiring up to three parameters, each of which may be a single-word variable, a list (array) of variables, or a literal containing up to 16 binary digits. One parameter may be specified in each address field in any desired sequence; none may be specified in the command code field.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION	COMMAND CODE	S/C	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1		TS	C	Z, CSC	X, +5	X, +0	
2	X,	TS	C	X, +3	Z, AU1	N, AU1	
3	X,	PR, S		[A ADDRESS PARAMETER]	[B ADDRESS PARAMETER]	[C ADDRESS PARAMETER]	
4	X,	CAC, S					
5	X,	SUBCALL				SUBR2	
6	X,	TX	S	X, +1	-	Z, CSC	
7	X,	RESERVE		1			

Figure A-5. Type 2 Calling Sequence

The coding of the type 2 calling sequence is given in Figure A-5. This routine performs the following functions:

Word 1 preserves (CSC) and resets CSC to X, +0, the address of the first out-of-sequence word in the calling sequence. Word 1 is located in sequence and replaces the call instruction.

Word 2 transfers the subroutine entry address to AU1, then resets the CSC to that address and gives control to the CSC to enter the subroutine.

Words 3 and 4 contain the parameter values stated in the call instruction. The "S" characters in the command code fields of these two words direct ARGUS to store the parameter values in these words according to a special format. Each

parameter which is expressed as a complex address is translated to machine form and placed in the corresponding address field of word 3. Each parameter which is expressed as a direct memory location address (with or without address modifier) or as a literal of up to 16 binary digits is translated to machine form and placed in the corresponding 16-bit group of word 4, and the address "N, AU2" is placed in the corresponding address field of word 3. The use of the "PR,S" and "CAC,S" pair allows an address in any valid address format to be made available to the subroutine. This pair also allows the use of a literal of up to 16 binary digits as a parameter.

Word 5, the SUBCALL constant, makes the subroutine entry address available to the calling sequence.

Word 6 restores the CSC to its preserved setting and returns control to the main coding. The final instruction of the subroutine exit section must give control to this word. The instruction following word 6 is selected by the sequencing counter specified in the S/C column of the call instruction.

Word 7 restores the temporary storage location for (CSC).

Each subroutine which utilizes a type 2 calling sequence requires an entry and an exit as linkages to and from the calling sequence. The following discussion illustrates a typical entry and exit for a subroutine which uses a type 2 calling sequence. The precise entry and exit prepared for a given subroutine depends upon the design of the call instruction and the types of parameters used. In every case, the parameter values must be presented in the form expected by the subroutine.

ENTRY: The following example illustrates an entry section based on three parameters in any valid address form, of which the A and B address fields contain one-word input parameters (arguments) and the C address field contains a one-word output parameter (result). Note that this is not necessarily the case when a type 2 calling sequence is used. The masks referenced by this entry coding are defined in the exit section (see below).

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION (1–10)	COMMAND CODE (11–22)	S/C (23)	A ADDRESS (24–37)	B ADDRESS (38–51)	C ADDRESS (52–65)	REMARKS (66–80)
1		TX	C	Z, CSH		CSHSAVE	
2		SS	C	N, CSH	AMASK	C, +4	
3		SWS, AMASK	C	N, CSH	36	C, +5	
4		SWS, ADES	C	N, CSH, I	1, L	C, +4	
5		SWS, ALLONES	C	N, CSH	32	Z, AU2	
6		TX	C	[A ADDR. PARAM.]		ASTORE	
7		SWS, ALLONES	C	N, CSH	16	Z, AU2	
8		TX	C	[B ADDR. PARAM]		BSTORE	

Word 1 preserves the contents of the CSH in a temporary location called CSHSAVE for exit purposes.

Word 2 substitutes the A address field of the "PR" word in the calling sequence into the A address field of word 6. Note that if the A address parameter is specified as a complex address, this parameter is transferred into word 6;

whereas, if the A address parameter is a direct memory location address, the address "N, AU2" is transferred into word 6.

Word 3 shifts and substitutes the B address field of the "PR" word into the A address field of word 8.

Word 4 shifts and substitutes the address designator bit corresponding to the address transferred in word 3. It also increments the CSH by 1 so that this register now contains the address of the "CAC" word in the calling sequence.

Word 5 transfers the A address field of the "CAC" word into AU2.

Word 6 transfers the contents of the location specified by the A address field of the "PR" word into a temporary storage location called ASTORE. Note that if the A address parameter is specified as a complex address, the contents of the location represented by this address are placed in ASTORE; whereas if the A address parameter is a direct memory location address, the contents of the location whose address is stored in AU2 are placed in ASTORE.

Words 7 and 8 position and transfer the contents of the location specified by the B address parameter into a temporary storage location called BSTORE.

If any of the parameters are literals or array locations, they must be treated specially and removed from the "PR" or "CAC" words of the calling sequence with coding other than the above. If the subroutine is independent and uses masks, the entry section must also preserve the contents of the MXR and set up the required mask group(s).

BODY: The body of the subroutine in this example uses the input parameters stored in ASTORE and BSTORE to perform the subroutine function, and stores the result in a temporary location such as ANSWER, where it is available to the exit section.

EXIT: The following example illustrates an exit section based on the same assumptions as the foregoing entry section, together with the masks required by both entry and exit. Again, this coding is merely representative of an exit that might be used with a type 2 calling sequence.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION (1–10)	COMMAND CODE (11–22)	S/C	A ADDRESS (24–37)	B ADDRESS (38–51)	C ADDRESS (52–65)	REMARKS (66–80)
1		TX	C	CSHSAVE		Z,CSH	
2		SS	C	N,CSH,1	CMASK	C,+2	
3		TX	C	N,CSH,2		Z,AU2	
4		TX	C	ANSWER		[C ADDR. PARAM.]	
5		TX	C	Z,CSH		Z,CSC	
6	S,ALLONES	DEC		GGGGGGGGGGG			
7	S,AMASK	DEC		100GGG			
8	S,ADES	DEC		1			
9	CMASK	DEC		040000000GGG			

Word 1 restores the preserved contents of the CSH (the address of the "PR" word in the calling sequence).

Word 2 substitutes the C address field of the "PR" word into the C address field of word 4. The contents of the CSH are incremented by 1 so that this register now contains the address of the "CAC" word in the calling sequence.

Word 3 transfers the "CAC" word (the word whose address is stored in the CSH) into AU2. Note that the low-order 16 bits of the "CAC" word are now stored in AU2. The CSH is incremented by 2 to form the address of the calling sequence exit instruction.

Word 4 transfers the contents of location ANSWER to the location specified in the C address field of word 4 (the C address parameter of the call instruction).

Word 5 transfers the contents of the CSH into the CSC and gives control to this counter to return control to the calling sequence exit instruction.

Special Calling Sequences

The standard calling sequences (type 1 and type 2) can be generated by ARGUS to handle many common forms of subroutines. Any subroutine which cannot conveniently use a standard calling sequence requires a special calling sequence which is coded as a macro routine. For example, a subroutine which requires more than three parameters must use a special calling sequence. In addition, a subroutine which does not require the flexibility inherent in the standard calling sequences (such as a one-parameter subroutine) can use a special calling sequence to advantage. Macro routines which are designed to serve as subroutine calling sequences have the same properties as other macro routines.

A special calling sequence must perform the same functions as a generated calling sequence; i.e., it must provide linkage with the subroutine and handle the parameters specified in the call instruction. The entry and exit sections of the subroutine must contain the coding to obtain the parameters and make the result information available to the main program. Since this coding is entirely dependent upon the macro routine which serves as calling sequence, examples are not provided.

Figures A-6 and A-7 show two macro routines designed to serve as special calling sequences. The routine CALLMAC (Figure A-6) is a special calling sequence designed to handle a subroutine with many parameters. The routine DBLSUM (Figure A-7) is a special calling sequence designed for efficient handling of a one-parameter subroutine. Figure A-7 shows both the generalized coding for the macro routine and the routine after insertion of the parameter values specified in the sample macro instruction.

Honeywell

Electronic Data Processing

CALLMAC

ARGUS CODING FORM

PROBLEM MACRO ROUTINE FOR CALLING SEQUENCE PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B. ADDRESS 51	52 C ADDRESS 65	REMARKS 66 LINE NUMBER 73	74 80
1		MACRODEF						
2		L,CALLMAC		TAG1/TAG2/TAG3	TAG4/BIN/PER	MASK/SR/SHIFT		
3		TS	C	Z,CSC	X,+9	X,+0		
4	X,	TS		X,+8	Z,AU1	N,AU1		
5	X,	PR,S		TAG1	TAG2	TAG3		
6	X,	CAC,S		TAG1	TAG2	TAG3		
7	X,	SWE,MASK		N,SR	A,SHIFT,L	N,SR,BIN		
8	X,	SPEC				TAG4+BIN		
9	X,	FXBIN		BIN				
10	X,	RF,PER						
11	X,	TX		X,+2		Z,CSC		
12	X,	SUBCALL				CALLMAC		
13	X,	RESERVE		1				
14		FINIS						
15								
16								
17								
18								
19								
20								

Figure A-6. Special Calling Sequence CALLMAC

Honeywell

Electronic Data Processing DBLSUM

ARGUS CODING FORM

PROBLEM MACRO ROUTINE FOR CALLING SEQUENCE PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	1 LOCATION 10	11 COMMAND CODE 22	S/C	24 A ADDRESS 37	38 B ADDRESS 51	52 C ADDRESS 65	REMARKS 66 LINE NUMBER 73	74 80
1		MACRODEF						
2		L,DBLSUM		LIST				
3		TS		C,+2	Z,AUI	N,AUI		
4		SPEC				LIST		
5		SUBCALL				DBLSUM		
6		FINIS						
7								
8								
9		AFTER INSERTING PARAMETER VALUES:						
10								
11		L,DBLSUM		DEBITS				
12								
13								
14		TS		C,+2	Z,AUI	N,AUI		
15		SPEC				DEBITS		
16		SUBCALL				DBLSUM		
17								
18								
19								
20								

Figure A-7. Special Calling Sequence DBLSUM

LAMP (Library Additions and Maintenance Program)

The library of routines, which is stored on the ARGUS symbolic program tape, is maintained by a program called LAMP. This program may be used to add a new routine to the library or to delete or modify one already in the library. The input to LAMP consists of the SPT (containing assembled programs and library routines) and a deck of punched cards or a tape containing punched card images. The input deck uses the following director cards to control the action of LAMP.

ARGUS CODING FORM

PROBLEM _____ PROGRAMMER _____ DATE _____ PAGE _____ OF _____

	LOCATION (1–10)	COMMAND CODE (11–22)	S/C	A ADDRESS (24–37)	B ADDRESS (38–51)	C ADDRESS (52–65)	REMARKS (66–73)	(74–80)
1		LAMP						
2		MACRODEF						
3		FINIS						
4		NEWSUB, g	d	PROGRAM NAME	SEGMENT NAME	SUBROUTINE NAME		
5		ELIMMAC		MACRO NAME				
6		ELIMSUB		SUBROUTINE NAME	MACRO NAME			
7		ENDLAMP						

LAMP: This director card precedes and identifies the LAMP input deck. The only significant information is the word LAMP punched in the command code field.

ENDLAMP: This director card signals the end of the LAMP input deck. The only significant information is the word ENDLAMP punched in the command code field.

Macro Routine Processing

The MACRODEF and FINIS directors are used to add a macro routine to the library. The card batch for a macro routine must contain the following cards, in the sequence in which they are listed:

1. A MACRODEF director card;
2. A master macro instruction card;
3. The cards containing the macrocoding; and
4. A FINIS director card.

If a macro routine is included in the LAMP input deck in this form, it is added to the library of routines in ARGUS input language. If a macro routine is included in the input to a program being assembled, the routine is handled as a programmer macro routine (see Section XIII) and is not added to the library.

MACRODEF: This director card precedes and identifies each macro routine in the input deck. The only significant information is the word MACRODEF punched in the command code field

and the contents of the S/C column. This column may direct LAMP to check the identity of all cards in the macro routine, to order the macrocoding cards by serial number, or both. Permissible characters in the S/C column are "I", "S", "B", or blank, and the resulting action is identical to the use of the MACRODEF instruction with a programmer macro routine (as described on page 84). In addition, if identity checking is requested for a given routine and any card within that routine fails the identity check, the routine is not added to the library and a diagnostic printout is produced.

FINIS: This director card signals the end of a macro routine. The only significant information is the word FINIS in the command code field. A FINIS card must be followed by another director card.

ELIMMAC: This card directs LAMP to delete a macro routine from the library. The word ELIMMAC is punched in the command code field and the name of the routine to be deleted is punched in the A address field. The remaining fields are not used.

In order to modify a macro routine in the library, it is necessary to delete the existing routine and add the new version, just as if it were an entirely new routine. The card deck for the new version must be complete with MACRODEF director, master macro instruction, macrocoding, and FINIS director. LAMP deletes the old version and adds the new version in two separate operations, always performing the deletion first, regardless of the relative positions of the ELIMMAC director and the new version in the input deck.

Subroutine Processing

Before a subroutine can be added to the library, it must be assembled and appear as a segment or as a complete program on the symbolic program tape. The directors NEWSUB and ELIMSUB are used to add subroutines to the library and to delete subroutines from the library, respectively.

NEWSUB: This card directs LAMP to add a new subroutine to the library and provides the information required to accomplish this. If the subroutine has been assembled as a single program segment, the program and segment names are punched in the A and B address fields, respectively, of the NEWSUB director, just as they appear on the symbolic program tape. If the subroutine has been segmented for assembly, the B address field is left blank and LAMP combines the segments to form a subroutine. In either case, the C address field must contain a name of up to eight alphanumeric characters by which the subroutine is to be identified in the library. A subroutine name must include at least one non-numeric character.

If the command code NEWSUB is followed by a comma and a digit "1" or "2", LAMP automatically generates a standard calling sequence of the specified type (see page 116) and adds this calling sequence to the macro routine portion of the library under the same name as the subroutine. If no information follows the command code, a macro routine should be added to the library for use as a special calling sequence. This routine must be in standard macro routine format with MACRODEF and FINIS directors and master macro instruction. It may or may not have the same name as the subroutine with which it is to be used.

The S/C column of the NEWSUB director must contain either a "D" or an "I" to indicate whether the subroutine is dependent upon or independent of the object program mask groups, respectively. A blank in this column is an error and results in a diagnostic comment. If the subroutine has been segmented for assembly, it must be specified as an independent subroutine.

ELIMSUB: This card directs LAMP to delete the subroutine named in the A address field from the library. The B address field may contain the name of a macro routine (usually the calling sequence for the subroutine to be deleted). In this case, both the subroutine and the macro routine are deleted. Otherwise LAMP deletes only the subroutine.

Before a subroutine in the library can be modified, the new version must be assembled and placed on the symbolic program tape. LAMP then requires an ELIMSUB director to delete the existing subroutine from the library, plus a NEWSUB director to add the new version. Since deletion is always performed before addition, the sequence of these directors within the input deck is irrelevant.

Output from LAMP

LAMP uses the same equipment configuration as the updating run (see page 91). The principal output from LAMP is the new symbolic program tape containing the updated library. In addition, LAMP produces a printed table of contents for the updated library, listing all macro routines and all subroutines in the library under separate headings. The printed output from LAMP also includes appropriate diagnostic comments if the following programming errors are detected in the input deck:

1. An illegal command code;
2. Card sequence error (i.e., violation of sequencing rules within a macro routine);
3. Illegal character;
4. Identity check failure;
5. The name of a routine exceeds eight characters in length;

6. A line number exceeds 2047;

7. Macro routine name duplicates name already in macro routine library;

(If any of the above errors is detected within a macro routine, the routine is not added to the library.)

8. Subroutine name duplicates name already in subroutine library;

9. A blank S/C column in a NEWSUB director (i.e., the subroutine is not designated either dependent or independent);

10. A subroutine contains errors which are unacceptable to LAMP:

11. A subroutine contains a reference to a special register group indicator;

12. A dependent subroutine contains more than 32 field masks or more than 64 shift masks;

13. A dependent subroutine contains a type "B" mask;

14. A dependent subroutine contains a reference to a mask as an operand;

(If any of the above errors 8-14 is detected, the subroutine is not added to the library.)

15. Illegal calling sequence number in a NEWSUB director. (If this error is detected, the subroutine is added to the library but no calling sequence is generated.);

16. The subroutine named on an ELIMSUB director is not in the library;

17. The macro routine named on an ELIMMAC director or on an ELIMSUB director is not in the library;

18. A SUBCALL constant in either a new or an existing macro routine names a subroutine which is not in the library (information only, no action by LAMP).

APPENDIX B

SYMBOLIC PROGRAM TAPE LAYOUT

The over-all layout of the symbolic program tape (SPT) is shown schematically in Figure B-1. The symbolic program tape contains a file of object programs in both symbolic ARGUS language and machine language. The retention of the original input language after assembly allows object program modification in ARGUS language, reassembly of existing programs during the updating run, and reproduction of assembled programs in ARGUS language. The object programs in the symbolic file also contain test data and debugging pseudo instructions (derails) for use by the Program Test System and information generated during assembly for use by other systems programs. In addition to the symbolic program file, the SPT also contains the file of ARGUS systems programs (Assembly, LAMP, the Program Test System, Executive, the library of routines, etc.) and a systems program loader. All programs in the systems file are stored in machine language.

Tape Label Record

The tape label record on the SPT is used to identify the tape. It also contains a bootstrap routine and a directory of the symbolic program tape. The bootstrap can be activated from the console to initiate loading of any of the systems programs. The directory lists the names of all programs and segments in the symbolic file, in the order in which they appear on tape. It is used to sort the ARGUS input deck into SPT order.

Loader

The two records following the tape label comprise the systems program loader. To load and execute a systems program from the SPT, the operator activates the bootstrap routine and types in the name of the desired program. The bootstrap loads the loader, which in turn loads and gives control to the program requested.

Systems Program File

The systems program file contains all of the programs comprising the ARGUS system in machine language. Following the begin file identification record, each systems program is preceded by a begin program identification record and a record of control information for use in operating the program. A systems program may include up to 64 segments. Each segment is headed by a begin segment identification record and consists of machine words and control information for loading the segment.

Symbolic Program File

A second begin file identification record separates the symbolic program file from the systems file. The elements of each object program are in the order which is established when ARGUS sorts the input card deck (see page 90). All object program modifications (insertions, deletions, and replacements) from previous updating runs are incorporated in their proper places. Program and segment directors are represented by begin program and begin segment identification records, respectively. Each test data card is represented by a two-word item containing the test data word and a control word. Each main coding word and each derail is represented by a variable-length symbolic item containing the original symbolic word, the assembled machine word, the absolute assignment of the word in memory, and the relocation information required by Executive.

A symbolic object program also contains the following items of information generated during the assembly process:

1. The RES table is a list of certain symbolic tags appearing in the program. It is used only during the reassembly process and is not of concern to the programmer.
2. The link tag table contains one item for each link tag used in the program. It is used by Executive during the relocation process.
3. The memory map is a list of all symbolic tags used in the program and their absolute assignments. It is used by the Program Test System to re-create ARGUS language from machine language.
4. The binary relocatable information consists of a two-word item for each machine word which is not represented by a symbolic item, i.e., subroutine words and generated masks.

Each program is followed by an end program identification record containing information about the equipment complement required by the program.

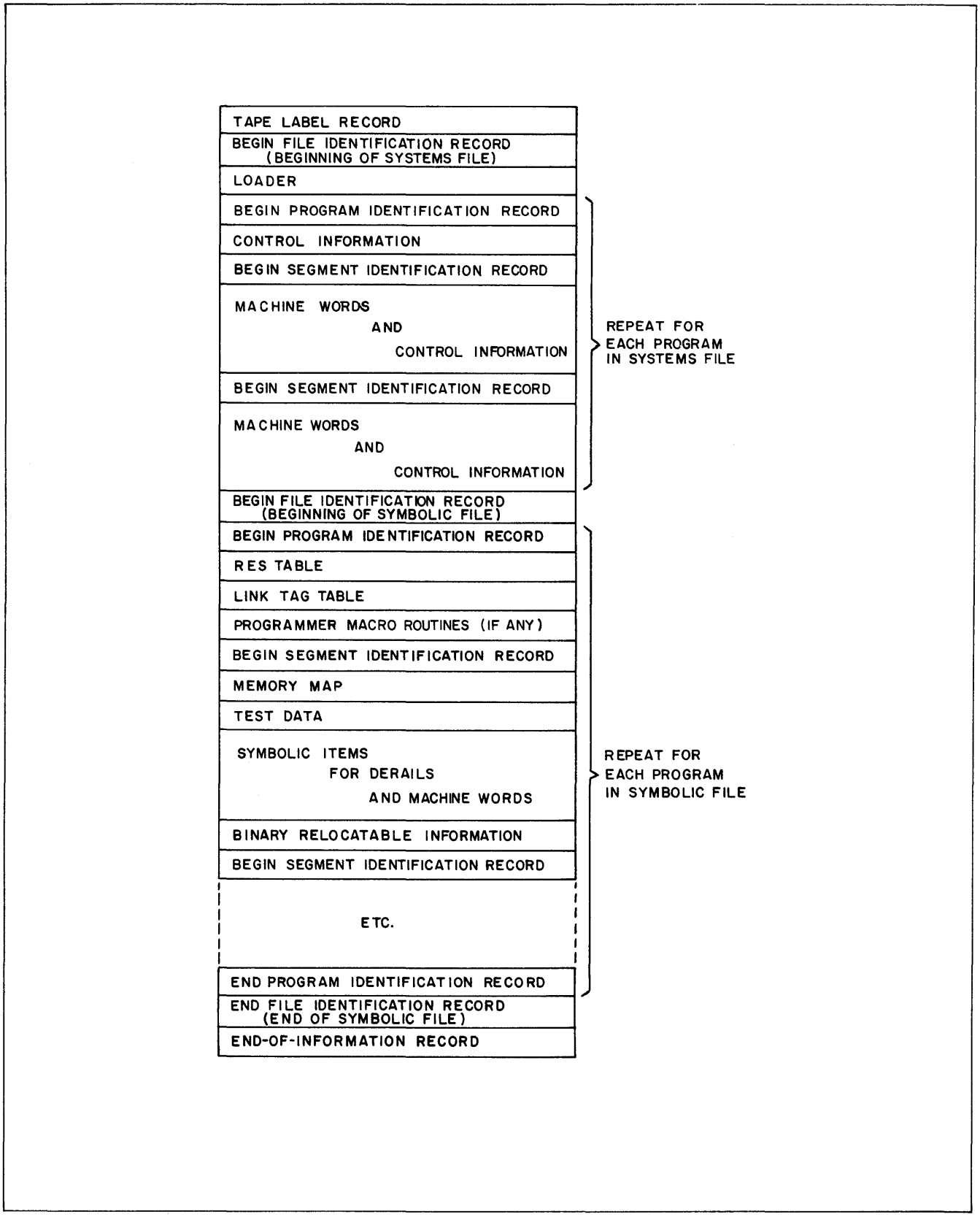


Figure B-1. Over-all Layout of the Symbolic Program Tape

APPENDIX C

ASSEMBLY EQUIPMENT CONFIGURATION CODE

The assembly equipment configuration code specifies the type and amount of equipment which ARGUS may assume to be available to a program being assembled. The equipment array which is available to a specific program is described by punching this code in the C address field of the PROGRAM card (see page 86). If this code is punched in the C address field of an ARGUS card (page 81), it describes a standard array which is available to all programs being assembled or reassembled during the current updating run for which individual arrays are not described. If no array is described on the ARGUS card, the standard array is assumed to be the configuration of the system on which the updating run is being performed. The equipment array actually required by the assembled program (known as the program configuration) may be either identical to or a subset of the assembly configuration. The Assembly Program checks the equipment requirements of the object program and produces a diagnostic comment if the program requires equipment which is not available.

The assembly configuration code consists of 10 characters punched in the C address field of the PROGRAM card or the ARGUS card. All 10 characters must be specified each time that the code is punched, since ARGUS automatically justifies the information to the left, suppressing any included spaces. If more than 10 non-space characters are punched, the first (left-most) 10 such characters are used and the remainder are ignored. The first eight characters in the configuration code correspond, respectively, to the eight input-output channel pairs in every Honeywell 800 system. Each of these characters may be a number from 0 to 8 or a letter from A to O, and is interpreted according to the following table. The ninth character must be a comma and the tenth a number from 1 to 4 which specifies the number of main memory modules (4096 words) in the available configuration.

Character	Significance
0 (zero)	No equipment is connected with the corresponding pair of channels.
1-8	A tape control is connected with the corresponding channel pair and the indicated number of tape units are attached, in turn, to this control unit. Any address up to and including the alphabetic code for this number is legal, since tape units must be plugged to consecutive hubs starting with 1.
A	A card reader is connected to the input channel; no equipment is connected to the output channel.

Character	Significance
B	No equipment is connected to the input channel; a printer is connected to the output channel.
C	No equipment is connected to the input channel; a card punch is connected to the output channel.

Note: If one of the letters A through C is punched, the indicated equipment may be connected by means of either a peripheral control or a multiple terminal unit control.

D	A card reader is connected to the input channel; a printer is connected to the output channel.
E	A card reader is connected to the input channel; a card punch is connected to the output channel.
F	No equipment is connected to the input channel; a printer and a card punch are connected to the output channel.
G	A card reader is connected to the input channel; a printer and a card punch are connected to the output channel.

Note: If one of the letters D through G is punched, the indicated equipment is connected by means of a multiple terminal unit control.

H	A card reader is connected to the input channel; a printer is connected to the output channel.
I	A card reader is connected to the input channel; a card punch is connected to the output channel.
J	A paper tape reader is connected to the input channel; no equipment is connected to the output channel.
K	No equipment is connected to the input channel; a paper tape punch is connected to the output channel.
L	A card reader is connected to the input channel; a paper tape punch is connected to the output channel.
M	A paper tape reader is connected to the input channel; a printer is connected to the output channel.
N	A paper tape reader is connected to the input channel; a card punch is connected to the output channel.
O (letter)	A paper tape reader is connected to the input channel; a paper tape punch is connected to the output channel.

Note: If one of the letters H through O is punched, all equipment on these channels is connected by means of individual peripheral controls.

APPENDIX D

TAPE, FILE, AND RECORD IDENTIFICATION

All of the ARGUS systems programs, as well as all 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 the end-of-information record define, respectively, the beginning and end of useful information on the tape. Each file or program on a 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 accounting procedures based upon the identification of the physical reel. On all ARGUS systems program tapes, the tape label record normally contains other information such as a bootstrap routine and/or a directory.

The maximum number of words in a tape label record is 2048. In the case of a data file or a 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. The tape label may be skipped by reading it into the stopper location; however, if it is to be rewritten, the contents of the first three words must be preserved for inclusion in the new tape label.

Word 1	The banner word in a tape label record has the octal configuration 6004 xxxx 0020 xxxx. The first four digits represent control information to bypass the tape label on a tape which is to be printed or punched. 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 138 for the binary configuration of a banner word.)

*Word 2	Tape Identification
*Word 3	Unspecified (contents preserved by ARGUS)
Words 4 to 11	Unspecified (may be used without restriction)
Words 12 to 12+n-1	Bootstrap Routine (systems program tapes only)
Words 12+n to 12+n+m-1	Program Directory (symbolic program tape only)
Words 12+n+m to 12+n+m+1	Orthowords
Word 12+n+m+2	End-of-Record Word

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 page 138).
Word 2	Name of File or Program (eight alphabetic characters)
Word 3	Reel Number (two low-order decimal digits) if 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 six decimal digits in the form year (two digits), month (two digits), day (two digits).
Words 5 to 5+n-1	File Parameters: Relocation information in program identification records; Sort parameters in file identification records (see below).
Words 5+n to 5+n+1	Orthowords
Word 5+n+2	End-of-Record Word

If a data file is to be sorted by an ARGUS generated sort or collate routine, words 5 to 9 of the file identification records should contain the following parameters, unless these parameters are supplied by means of "own coding".

Word 5	Digits 1-3	Number of items per record (1-250)
	Digits 4-6	Number of words per item (1-250)
	Digit 7	Fixed-length (0) or variable-length (1) records
	Digit 8	Banner words present (0) or missing (1) in data records
	Digits 9-12	Not used

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

Word 6	Digits 1-3	1st key position (word 1-250)
	Digits 4-6	2nd key position (word 1-250)
	Digits 7-9	3rd key position (word 1-250)
	Digit 10	Keys not masked (0) or masked (1)
	Digits 11-12	Not used
Word 7		Mask for 1st key
Word 8		Mask for 2nd key
Word 9		Mask for 3rd key

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 page 138).
Word 2	Name of Program (eight alphanumeric characters)
Word 3	Name of Segment (seven high-order alphanumeric characters)
Word 4	Date Obsolete and Date Written (begin segment records only). Each date comprises two decimal digits for year, two digits for month, and two digits for day.
Words 5 to 5+n-1	Relocation Information
Words 5+n to 5+n+1	Orthowords
Word 5+n+2	End-of-Record Word

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 page 138).
Words 2-4	Unspecified
Words 5-6	Orthowords
Word 7	End-of-Record Word

Banner Words

The first word of every record is a banner word which should contain a record count in bit positions 33 through 48. This 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, including Executive, 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 bit 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 a tape label record, bit position 1 should contain a 1, which causes the peripheral control to ignore the remaining control bits (2-30) if the tape is printed or punched. The printer, for example, prints and then skips to the head of form.

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 (used only on symbolic program tape to identify the boundaries of the two program files)

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.

Summary

Some of the above conventions are required for tapes to be used with Honeywell automatic programming aids and some are optional. As noted above, the restart provisions of the Executive System require the inclusion of banner words in all records. Likewise, the ARGUS generated sort and collate routines assume the presence of parameters in words 5 through 9 of the begin file identification records (as noted above), unless this information is provided by means of "own coding". These routines also require that each file contain an end file identification record which specifies the same file name as the begin file identification record. In addition, the library tape-handling routines assume the presence of banner words in all records, plus tape label records, end-of-information records, and beginning-of-file, program, and segment identification records as appropriate. The use of end program and segment identification records is optional. Such records may be included wherever their use facilitates processing.

APPENDIX E

HONEYWELL 800 MACHINE INSTRUCTIONS

The following pages contain a tabular summary of the Honeywell 800 complement of machine instructions, grouped according to the major instruction categories. The mnemonic ARGUS operation code and the basic time of each instruction are given, together with a brief description of the function(s) performed. The machine instructions are described in greater detail in the Honeywell 800 Programmers' Reference Manual. In particular, the basic instruction times shown in this Appendix may be affected by masking, indexed and indirect addressing, and inactive addressing. A detailed listing of instruction times considering all of these factors is presented in Appendix C of the Reference Manual.

Mnemonic Operation Code	Description	Time in Memory Cycles[1]
	GENERAL INSTRUCTIONS	
BA[2]	Binary Add algebraically (A) to (B). Store sum in C. If overflow occurs, transfer this instruction to the address stored in the unprogrammed transfer register (UTR) and take the next instruction from (UTR) + 8 if the sequence counter selected this instruction; transfer this instruction to (UTR) + 1 and take the next instruction from (UTR) + 9 if the cosequence counter selected this instruction. The sign of either operand is positive if the sign digit contains any "1" bit. The sign of the sum is 0000 if negative, 1111 if positive.	4
DA[2]	Decimal Add algebraically (A) to (B). Store sum in C. Otherwise same as BA.	4
BS[2]	Binary Subtract algebraically (B) from (A). Store result in C. Observe same overflow and sign conventions as in BA.	4
DS[2]	Decimal Subtract algebraically (B) from (A). Store result in C. Otherwise same as BS.	4
BM	Binary Multiply (A) by (B). Store high-order product with proper sign in C and accumulator, low-order product with proper sign in low-order product register. Product signs are 0000 if negative, 1111 if positive.	33
DM	Decimal Multiply (A) by (B). Store high-order and low-order products as in BM with same sign conventions.	27
WA[2]	Word Add. Binary add absolute value of (A) to absolute value of (B), considered as unsigned 48-bit numbers. Store 48-bit result in C. Observe same overflow conventions as in BA.	4
WD[2]	Word Difference. Binary subtract absolute value of (B) from absolute value of (A). Otherwise identical to WA.	4

Mnemonic Operation Code	Description	Time in Memory Cycles[1]
	GENERAL INSTRUCTIONS (cont)	
HA[2]	Half Add. Binary add (A) and (B) without carry. Store result in C. (A) and (B) are unsigned 48-bit numbers. Bits of C are 0 wherever corresponding bits of (A) and (B) are identical, 1 wherever they are different.	4
SM[2]	Superimpose (A) and (B). Store result in C. Bits of C are 0 wherever bits of both (A) and (B) are 0, 1 everywhere else.	4
BT	Binary Accumulate. Clear the accumulator. Transfer (A) to the accumulator. Perform this transfer B times (0-63 times), regarding transferred words as signed 44-bit numbers. Add absolute values of transferred words. Note that if A is an indirect address with a non-zero increment, B different numbers are accumulated. Store result with sign of the first word transferred in C. Observe same overflow conventions as in BA.	$3 + n^{(3)}$
DT	Decimal Accumulate. Same as binary accumulate except that transferred words are regarded as signed 11-digit decimal numbers.	$3 + n^{(3)}$
SS	Substitute. Using (B) as a mask, transfer (A) to C, protecting unmasked portions of C. Note that this is never a field instruction.	5
EX	Extract. Using (B) as a mask, transfer (A) to C without protecting unmasked portions of C. Result is 1 wherever bits (A) and (B) are both 1, 0 everywhere else. Note that this is never a field instruction.	5
TX[2]	Transfer (A) to C. Ignore B.	3
TS[2]	Transfer (A) to B. Change specified counter to +C, unless C is inactive.	4
TN	N-Word Transfer. Transfer B words from consecutive locations starting at A to consecutive locations starting at C. From 0 to 63 words may be transferred.	$5 + 2n^{(3)}$
IT	Item Transfer. Substitute end-of-item symbol[4] for high-order 32 bits of (B), clearing low-order 16 bits of (B) to all zeros. Transfer words from consecutive locations starting with A to consecutive locations starting with C until an end-of-item symbol is transferred. Upon completion, AU1 contains A + n, AU2 contains C + n.	$7 + 2n^{(3)}$
RT	Record Transfer. Store end-of-record word[5] in B. Transfer words from consecutive locations starting with A to consecutive locations starting with C until an end-of-record word is transferred. Otherwise same as Item Transfer.	$7 + 2n^{(3)}$

Mnemonic Operation Code	Description	Time in Memory Cycles[1]
	GENERAL INSTRUCTIONS (cont)	
NN[2]	Inequality Comparison, Numeric. Compare algebraically (A) and (B). If (A) ≠ (B), change specified counter to +C. Plus 0 equals minus 0.	4
NA[2]	Inequality Comparison, Alphabetic. Same as NN except that absolute values of (A) and (B) including sign positions are compared. Plus 0 is not equal to minus 0.	4
LN[2]	Less Than Or Equal Comparison, Numeric. Compare algebraically (A) and (B). If (A) ≤ (B), change specified counter to +C. Plus 0 equals minus 0.	4
LA[2]	Less Than Or Equal Comparison, Alphabetic. Same as LN except that absolute values of (A) and (B) including sign positions are compared. Plus 0 is greater than minus 0.	4
PR	Proceed (no operation).	2
CC	Compute Orthotronic Count. Write a generated end-of-record word[5] in C. Compute orthotronic count from A to end-of-record word. Store orthoword 1 in C, orthoword 2 in C + 1. Write an end-of-record word in C + 2. Store +B in AU2. If B is inactive, control is not changed for distributed item handling. If B is active, end-of-item words[4] are sensed and control is changed for distributed item handling.	11 + n[(3)]
CP[2]	Check Parity. Test (A) for correct parity. Place (A) with correct check bits in B. If (B) differs from (A), change specified counter to +C.	4
MT	Multiple Transfer. Transfer (A) to C. Perform this instruction B times (0-63 times). Note that if A and C are indirect addresses with non-zero increments, B different transfers will be performed.	1 + 2n[(3)]
MPC	Control Program. Ignore A. Place (PCR) in the location specified by C. Then alter the bits of PCR specified by bits 5 through 12 of B, using bits 1 through 4 of B to define how the bits are altered. If B address memory designator bit is 1, hunt for next program in demand. Otherwise, do not hunt.	4
	SHIFT INSTRUCTIONS	
SPS	Shift Preserving Sign and Substitute. Shift end-around excluding sign (A) as directed by B. Mask the result and store in C (protected). B contains three parameters to direct the shift: a character representing the number of bits per position shifted, (B_1); the number of positions shifted (B_2); and the direction of shift (B_3).	5 + k[(6)]

Mnemonic Operation Code	Description	Time in Memory Cycles[1]
	SHIFT INSTRUCTIONS (cont)	
SPE	Shift Preserving Sign and Extract. Same as SPS except that the unmasked portions of C are unprotected (cleared to all 0 bits).	$5 + k^{(6)}$
SWS	Shift Word and Substitute. Shift end-around including sign (A) as directed by B. Mask the result and store in C (protected). Otherwise same as SPS.	$5 + k^{(6)}$
SWE	Shift Word and Extract. Same as SWS except that the unmasked portions of C are unprotected (cleared to all 0 bits).	$5 + k^{(6)}$
SSL	Shift and Select. Shift end-around including sign (A) as directed by B. Binary add under mask control the absolute value of result to C to form C'. Change the specified counter to +C'. The shift operation is specified as in SPS. The mask must not cause more than 11 low-order bits to be added to C.	$6 + k^{(6)}$
	SIMULATOR INSTRUCTION	
S	Simulator. Form a memory location address (direct or indexed) from the low-order 11 bits of the command code and store this instruction in the location thus specified. Change the cosequence counter to select the next instruction from the next higher address.	7
	PERIPHERAL INSTRUCTIONS	
WF, XX	Write Forward on peripheral device XX the contents of consecutive memory locations from A through the end-of-record word[5]. (If device XX is a terminal output unit, print one line or punch one card.) Set write address counter (WAC) to +A. If B is <u>not</u> inactive, the distributed write address counter (DWAC) is set to +B and end-of-item symbols[4] are sensed for distributed writing of a multi-item record. If B is inactive, end-of-item symbols are <u>not</u> sensed. Change the source counter to +C unless C is inactive (S/C subfield is not used). If end of tape is sensed, transfer this instruction to the contents of the unprogrammed transfer register (UTR) if the sequence counter selected this instruction or to (UTR) + 1 if the cosequence counter selected this instruction and take the next instruction from (UTR) + 4 or (UTR) + 5. If an error was detected during the last previous write to peripheral device XX, reset the error, do not perform this instruction. Instead, transfer this instruction to (UTR) or (UTR) + 1 and take next instruction from (UTR) + 6 or (UTR) + 7. This instruction is interlocked against device XX and the associated buffer.	5

Mnemonic Operation Code	Description	Time in Memory Cycles[1]
	PERIPHERAL INSTRUCTIONS (cont)	
RF, XX	Read Forward from peripheral device XX into consecutive memory locations starting with A through the end-of-record word. (If device XX is a card reader, read one card.) Set read address counter (RAC) to +A. If B is not inactive, the distributed read address counter (DRAC) is set to +B and end-of-item symbols[4] are sensed for distributed reading of a multi-item record. If B is inactive, end-of-item symbols are not sensed. Change the source counter to +C unless C is inactive (S/C subfield is not used). End-of-tape and parity-error rules are identical with write forward instruction. This instruction is interlocked against device XX and the associated buffer.	5
RB, XX	Read Backward from magnetic tape unit XX into consecutive memory locations starting with A. This instruction is otherwise identical to read forward except that the RAC is set to -A and, if B is not inactive, the DRAC is set to -B.	5
RW, XX	Rewind Tape on peripheral device XX to beginning. If already rewound, proceed. If A is active, interlock device XX against any further peripheral operations. B and C are ignored. If an error was detected during the last previous read or write for this tape, reset the error and perform the rewind.	2
	EXTENDED INSTRUCTIONS	
STOP	Transfer the contents of the program control register (PCR) to C, unless C is inactive. Ignore A and B. Stop the program in which this instruction appears. Hunt for another demand.	4
DOFF	Transfer (PCR) to C, unless C is inactive. Ignore A. Turn off up to seven programs specified by B. This instruction specifies whether or not to hunt for another demand.	4
DON	Transfer (PCR) to C, unless C is inactive. Ignore A. Turn on up to seven programs specified by B. This instruction specifies whether or not to hunt for another demand.	4
SCON	Transfer (PCR) to C, unless C is inactive. Ignore A. Turn control of up to seven programs specified by B over to their respective sequence counters. Turn on these programs. This instruction specifies whether or not to hunt for another demand.	4
CSCON	Transfer (PCR) to C, unless C is inactive. Ignore A. Turn control of up to seven programs specified by B over to their respective cosequence counters. Turn on these programs. This instruction specifies whether or not to hunt for another demand.	4
SPCR	Transfer (PCR) to C, unless C is inactive. Ignore A and B.	4

Mnemonic Operation Code	Description	Time in Memory Cycles[1]
	EXTENDED INSTRUCTIONS (cont)	
SPCR (cont)	This instruction specifies whether or not to hunt for another demand.	
PRA	Print (A) alphanumerically on the automatic typewriter specified in B. Change the specified counter to +C, unless C is inactive.	5
PRD	Print (A) hexadecimally on the automatic typewriter specified in B. Change the specified counter to +C, unless C is inactive.	5
PRO	Print (A) octally on the automatic typewriter specified in B. Change the specified counter to +C, unless C is inactive.	5
	SCIENTIFIC INSTRUCTIONS	
FBA	Floating Binary Add. Binary add algebraically (A) to (B). Deliver the result as a normalized floating-point number to C if C is active; retain the result in FLAC if C is inactive. If exponential underflow occurs, take next instruction from (UTR) + 12 or (UTR) + 13. If exponential overflow occurs, take next instruction from (UTR) + 14 or (UTR) + 15. If either occurs, store current instruction in (UTR) or (UTR) + 1.	7
FDA	Floating Decimal Add. Same as floating binary add with the word "binary" replaced by "decimal".	7
FBS	Floating Binary Subtract. Change the sign of the B operand and perform a floating binary add.	7
FDS	Floating Decimal Subtract. Same as floating binary subtract with the word "binary" replaced by "decimal".	7
FBAU	Floating Binary Add, Unnormalized. Same as floating binary add, except that the result is not normalized. A four-bit shift to the right is provided if necessary to compensate for mantissa overflow, but no compensating left shift occurs to renormalize a result with 0 in the most significant mantissa digit.	7
FDAU	Floating Decimal Add, Unnormalized. Same as floating binary add, unnormalized, with the word "binary" replaced by "decimal".	7
FBSU	Floating Binary Subtract, Unnormalized. Change the sign of the B operand and perform a floating binary add, unnormalized.	7
FDSU	Floating Decimal Subtract, Unnormalized. Same as floating binary subtract, unnormalized, with the word "binary" replaced by "decimal".	7

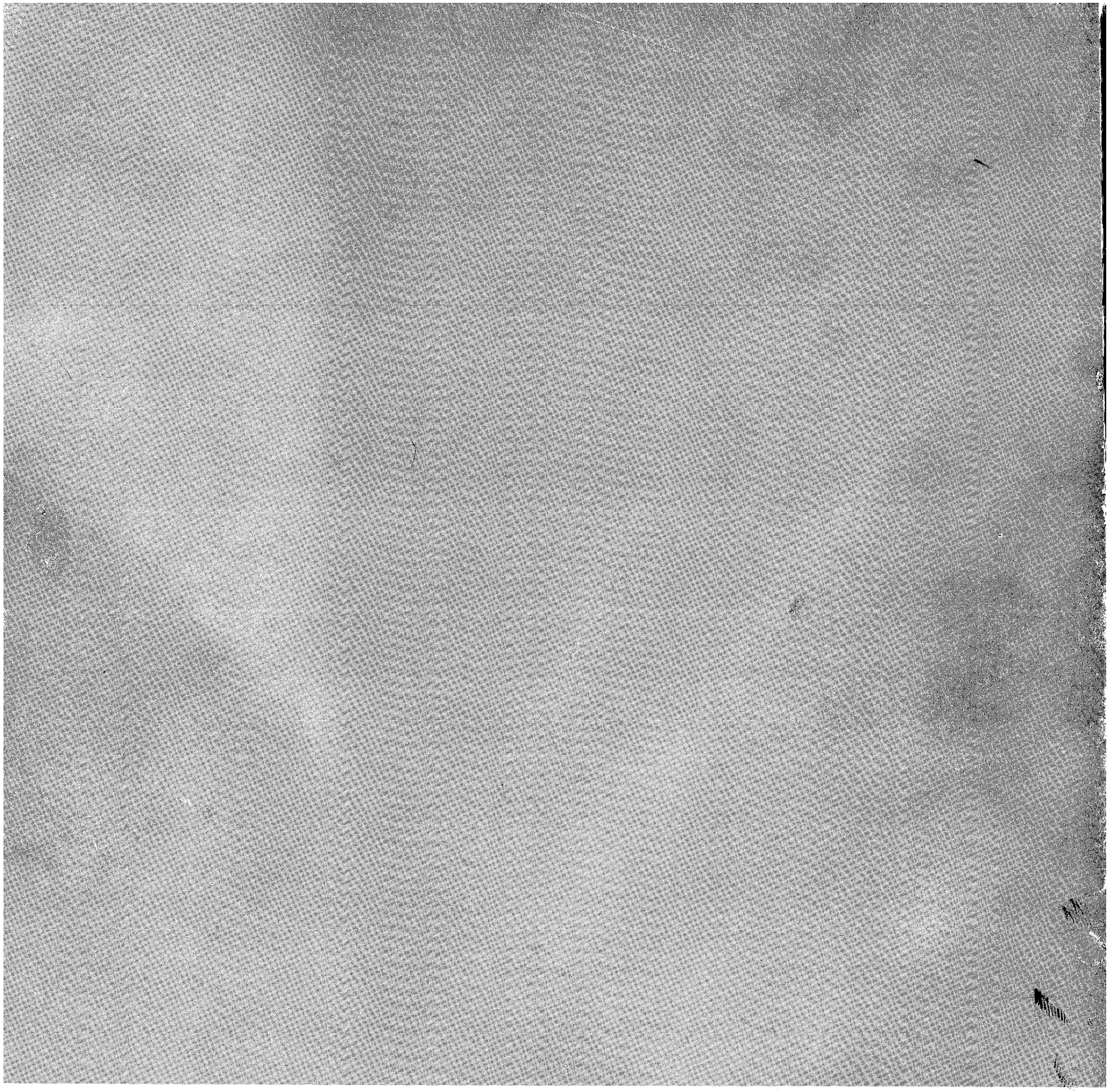
Mnemonic Operation Code	Description	Time in Memory Cycles[1]
	SCIENTIFIC INSTRUCTIONS (cont)	
FBAE	Floating Binary Add, Extended Precision. Form the normalized double-precision sum of (A) and (B). If C is inactive, retain the high-order and low-order parts in FLAC and FLOP. If C is active, deliver the high-order part to C and the contents of FLOP are unspecified. Sense for exponential overflow or underflow on the high-order result. If exponential underflow occurs on the low-order result, set the low-order underflow indicator.	10
FBSE	Floating Binary Subtract, Extended Precision. Change the sign of the B operand and perform a floating binary add, extended precision.	10
FBM	Floating Binary Multiply. Multiply (A) by (B). The resulting product is a normalized, double-precision, floating-point number whose high-order part is stored in C. If C is inactive, retain the high-order product in FLAC and store the low-order product in FLOP. Sense for exponential overflow or underflow on the high-order product. If exponential underflow occurs on the low-order product, set the low-order underflow indicator.	13.6
FDM	Floating Decimal Multiply. Same as floating binary multiply with the word "binary" replaced by "decimal".	40.5
FBD	Floating Binary Divide. Divide (B) by (A). Store the quotient in floating-point form in C. If C is inactive, retain the quotient and remainder in FLAC and FLOP. The quotient is normalized if the operands are normalized. The remainder is not normalized. Sense for exponential overflow or underflow in the quotient. Set the remainder underflow indicator if there is underflow in the remainder. If the divisor is unnormalized or 0, store this instruction in (UTR) or (UTR) + 1 and take the next instruction from (UTR) + 10 or (UTR) + 11.	67
FDD	Floating Decimal Divide. Same as floating binary divide with the word "binary" replaced by "decimal".	64
BD	Fixed Binary Divide. Divide (B) by (A), where both operands are considered as fixed-point binary numbers. If the absolute value of (B) equals or exceeds the absolute value of (A), take the next instruction from (UTR) + 10 or (UTR) + 11; the contents of C are unspecified. If C is active and the absolute value of (B) is less than the absolute value of (A), place the quotient in C. If C is inactive, retain the quotient and the remainder in FLAC and FLOP.	74
DD	Fixed Decimal Divide. Same as fixed binary divide with the word "binary" replaced by "decimal".	68
FLN	Normalized Less Than Comparison. Compare (A) with (B). If (A) $\leq$ (B), change the specified counter to +C. If (A) and (B) have different signs (bit 1), then the positive exceeds the negative. If both operands are positive, then the operand with larger	4

Mnemonic Operation Code	Description	Time in Memory Cycles[1]
	SCIENTIFIC INSTRUCTIONS (cont)	
FLN (cont)	exponent exceeds the other unless the exponents are alike, in which case the operand with larger mantissa exceeds the other. If both operands are negative, then the operand with smaller exponent exceeds the other unless the exponents are alike, in which case the operand with smaller mantissa exceeds the other.	
FNN	Normalized Inequality Comparison. Compare (A) with (B), including sign positions. If (A) ≠ (B), change the specified counter to +C. This is identical to the general instruction inequality comparison, alphabetic.	4
FFN	Fixed to Floating Normalize. Take the least significant 44 bits of (B) as a mantissa to be normalized. If C is active, store the normalized mantissa in the least significant 40 bits of C. If C is inactive, the FLAC and FLOP contain a normalized double-precision number whose high-order part is in FLAC and whose least significant 36 bits in FLOP are zeros. The exponent of the result is the binary sum of the exponent of (A) minus the amount of left shift plus the amount of right shift minus 1. The sign of (C) is the logical sum of the four sign bits of (B). Sense for exponential underflow in the high-order part. Set the low-order underflow indicator if exponential underflow occurs in the low-order part.	5
ULD	Multiple Unload. Place the contents of FLAC in A and the contents of FLOP in C. The B address must be inactive. If either the low-order underflow indicator or the remainder indicator is set when the instruction is initiated, take the next instruction from (UTR) + 12 or (UTR) + 13.	4

NOTES

1. One memory cycle equals six microseconds.
2. Instructions so designated are field instructions and may be executed under control of field masks.
3. n = number of words transferred, accumulated, or orthocounted.
4. An end-of-item symbol is a word whose high-order 32 bits are
   1010 1010 0000 0000 1110 1110 1110 1110
5. An end-of-record word has the configuration
   1010 1010 0000 0000 1110 1110 1110 1110 1101 1101 1101 1101
6. Values of k vary from 0 to 4, based on number of 16-, 4-, and 1-bit shifts required (see Reference Manual).

INDEX



Honeywell

Electronic Data Processing