DEFINE MNEMONICS MENU 7

The Define Mnemonics menu provides disassembly capability ranging from simple lookup tables to complete microprocessor disassembly. While the menu may be displayed on the monitor screen at any time, it is useful only if data acquisition modules are installed in the mainframe.

In this section you will find: **Page**

LIST OF ILLUSTRATIONS

DEFINE MNEMONICS MENU

Overview

The Define Mnemonics menu is used to set up disassembly tables for the State Table display. It is composed of three sub-menus which allow you to define mnemonics at two different levels: lookup tables for simple applications, and call tables for complete microprosessor disassembly.

Figure 7-1 provides a general overview of how the Define Mnemonics menu works with the other DAS menus.

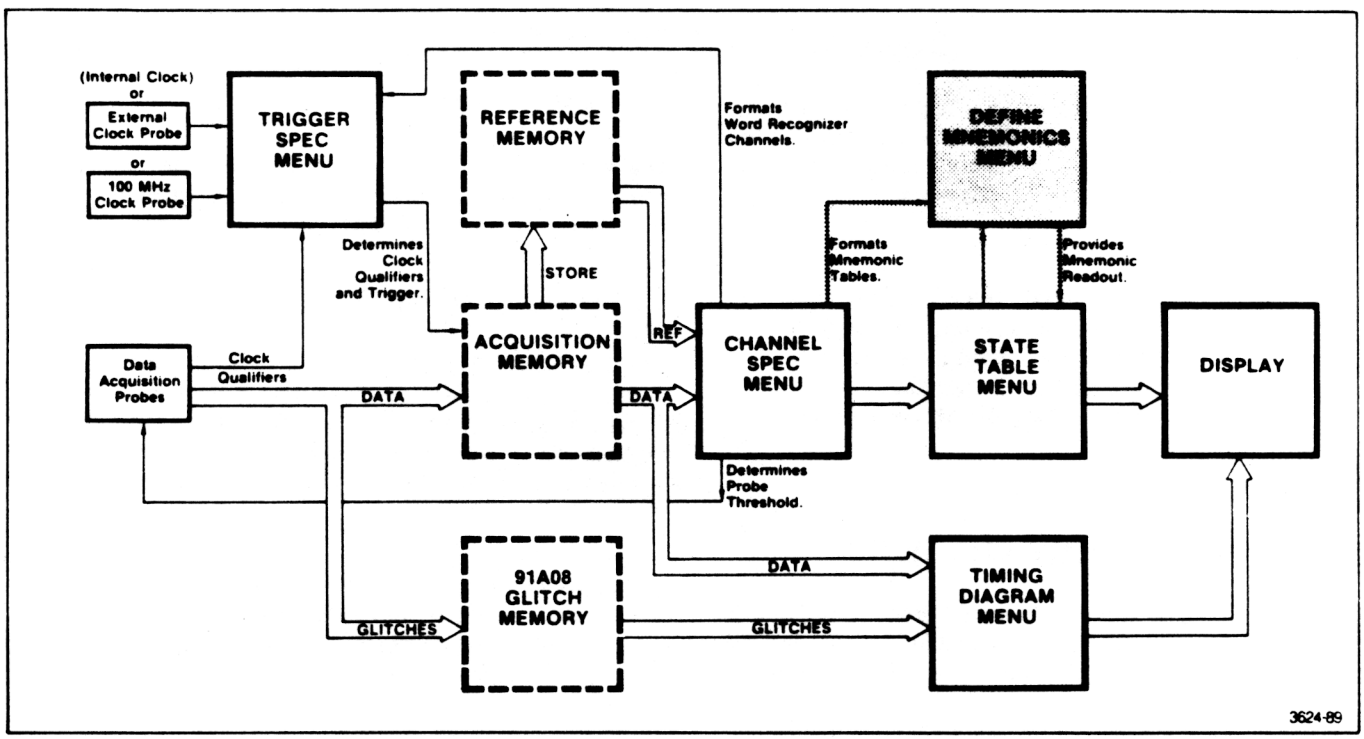


Figure 7-1. Functional overview of the Define Mnemonics menu. This menu reads the channel group information from the Channel Specification menu and provides the level of disassembly you select using the sub-menus. The acquired data and/or the associated mnemonics selected appear in the State Table display.

Sub-Menu Functions

The three Define Mnemonics sub-menus are called Table Entry, Display Setup, and Table Definition. These sub-menus allow you to use lookup tables for simple applications, or call tables for complete microprocessor disassembly.

Lookup Tables. The default configuration of the Define Mnemonics menu is set up for lookup tables. This configuration provides one mnemonic table for each channel group (A-F, 0-9). To use the default menu configuration, you enter values only in the Table Entry and Display Setup sub-menus.

To enter the Table Entry and Display Setup sub-menus, press the DEFINE MNEMONICS menu key.

Table Entry sub-menu – This sub-menu allows you to enter data values and associated mnemonics in disassembly tables.

Display Setup sub-menu – This sub-menu allows you to alter the default State Table's display format for acquired data and mnemonics.

To move between these two sub-menus, place the cursor in the MODE field and press the SELECT key.

Microprocessor Disassembly. For microprocessor disassembly applications, the Define Mnemonics menu can be reconfigured, letting you create call table setups that are personalized for your system under test. This capability is available through the Table Definition sub-menu.

To enter the Table Definition sub-menu, simultaneously press the SHIFT and DEFINE MNEMONICS keys.

Table Definition sub-menu — This sub-menu allows you to define new tables, to specify more than one channel group for input to tables, and to set up a call table structure. The call table structure lets you call one table from another, pass data bits from one table to another, and use special system calls.

To move from Table Definition to the Table Entry and the Display Setup sub-menus, press the DEFINE MNEMONICS menu key. To move directly to any other menu, press the appropriate menu key.

LOOKUP TABLE DISASSEMBLY

The following paragraphs describe the Table Entry sub-menu in its default configuration (i.e., with no entries made in the Table Definition sub-menu). The default configuration is intended for simple lookup table applications.

NOTE

Non-default configuration of the Table Entry sub-menu changes its appearance and use, and offers features intended for complete microprocessor disassembly. Details are provided later in this section under the title Complete Microprocessor Disassembly.

TABLE ENTRY SUB-MENU

To enter the Table Entry sub-menu, press the DEFINE MNEMONICS menu key and select the TABLE ENTRY mode (by pressing the SELECT key while in the MODE field).

The Table Entry sub-menu allows you to enter mnemonic definitions for disassembly tables. In its default configuration, the sub-menu provides one table for each channel group. You can enter up to 256 definitions in each table. The sub-menu displays one table at a time.

Figure 7-2 shows how the Table Entry sub-menu relates to the DAS Channel Specification menu and State Table display. The Table Entry sub-menu receives channel group and bit organization from the Channel Specification menu. Then, using the mnemonic definitions entered in its disassembly tables, it provides mnemonics for display on the State Table menu.

Figure 7-3 shows the Table Entry sub-menu's default display. Refer to the numbered call-outs in Figure 7-3 when reading the following paragraphs. The numbers are intended as a visual reference, and do not imply sequence of use.

(1) MODE Field

The MODE field is used to select between the Table Entry and Display Setup sub-menus. To move between these two sub-menus, press the SELECT key.

(2) TABLE NAME Field

The TABLE NAME field lets you specify which table you want to display on the screen. Only one table is displayed at a time. In the default menu configuration, there is one table defined for each of the 16 channel groups. The tables have the same names (A-F, 0-9) as their corresponding channel groups.

NOTE

You can also specify tables that you have defined in the Table Definition sub-menu. Definition of tables is explained in the Table Definiton Sub-Menu portion of this section.

To select a table for display:

Press the DON'T CARE key to clear the field, and use the DATA ENTRY keys to type the table's name.

If you have selected a valid table, the new table will be displayed on the sub-menu. Invalid tables are those that have no channels assigned, and those that have not been defined.

If you specify an undefined table and attempt to move the cursor out of the Table Name field, the error message TABLE HAS NOT BEEN DEFINED will appear on the second line of the screen.

If you select a defined table that has no channels assigned to it, the error message NO CHANNELS ASSIGNED will appear in place of the usual sequence, data, and display columns.

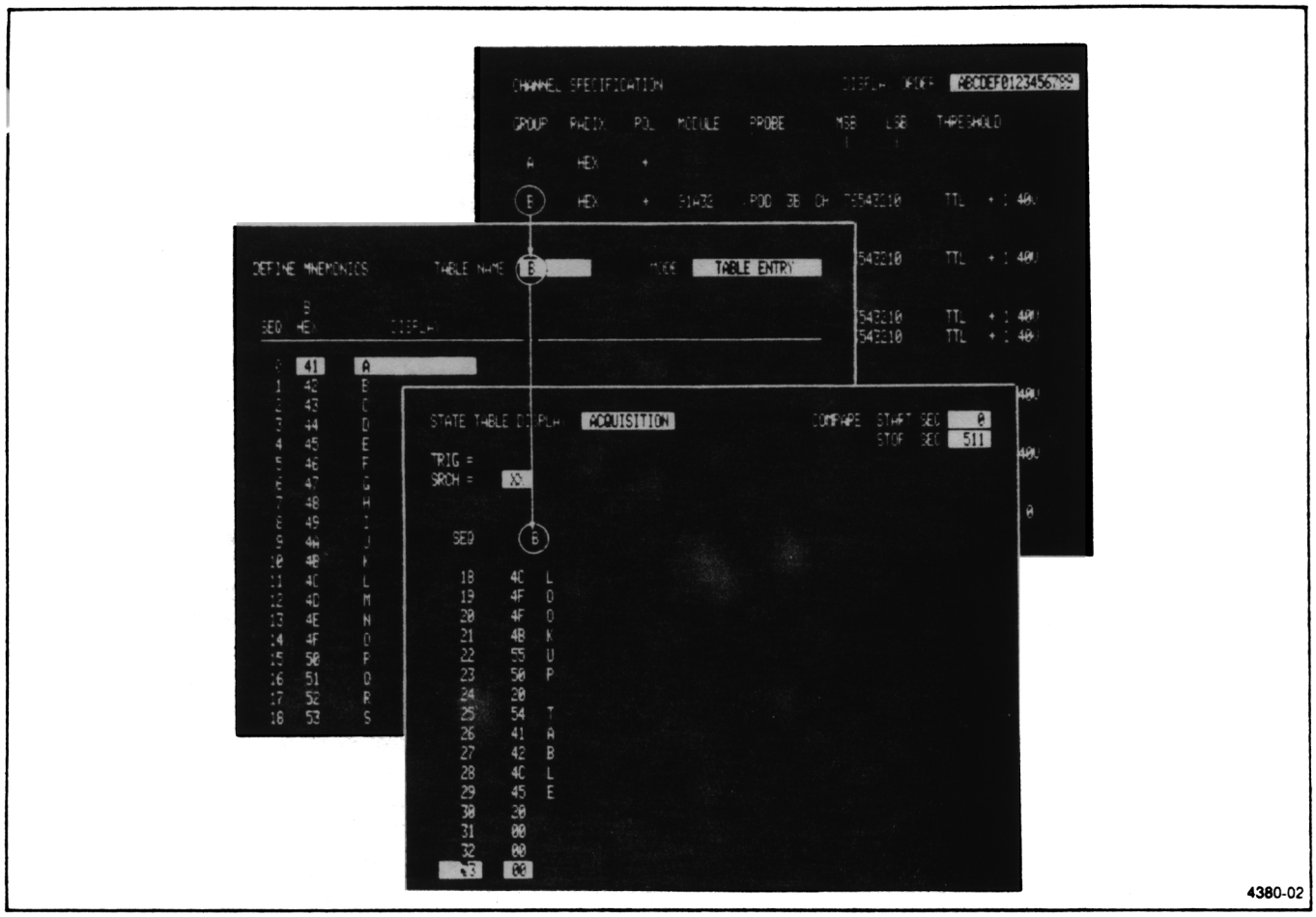


Figure 7-2. Relationship of Table Entry sub-menu to the Channel Specification and State Table menus. This simple ASCII lookup table shows how values entered in the Channel Specification menu and Table Entry sub-menu affect the data displayed in the State Table.

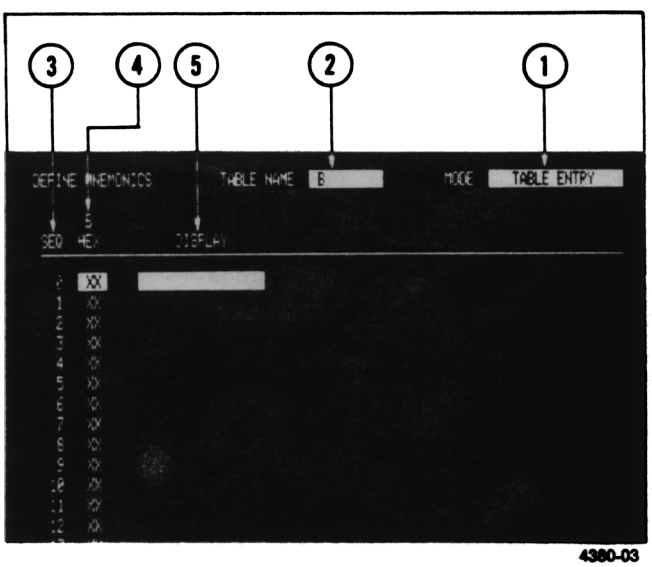


Figure 7-3. TABLE ENTRY sub-menu and its fields. This illustration shows the sub-menu in its default configuration. (When using the Table Definition sub-menu to define nested tables, additional fields will appear in this sub-menu. For details, refer to Table Definition Sub-Menu and Using Call Tables in the Complete Microprocessor Disassembly section.)

(3) SEQ (Sequence) Column and Table Editing

This column shows the sequence number of each entry in the disassembly table. A default sequence occupies one line of the display and consists of a value field and a DISPLAY field. Altogether, there are 256 sequences (0-255) available for each table that has one or more channels assigned to it. The screen displays 19 entries (lines) at a time.

To scroll through sequences:

Press the ∧ or ∨ key.

Adding and Deleting Sequences. Two editing functions are provided for the mnemonic tables:

To add a sequence:

1. Move the screen cursor to the sequence line where you want to add a new line. For example, SEQ 0:

   0 [10110010] [JMP]

2. Press the ADD LINE key.

   The DAS moves the values associated with SEQ 0 down one line, and inserts a new empty sequence at the cursor location:

   0 [XXXXXXXX] [ ]

   1 [10110010] [JMP]

To delete a sequence:

1. Move the screen cursor to the sequence you want to delete. For example, SEQ 0:

   0 [00000000] [ ]

   1 [10110010] [JMP]

2. Press the DEL LINE key.

   The DAS deletes that line from the table, and moves all following sequences up one line.

   0 [10110010] [JMP]

(4) VALUE Field

The VALUE field allows you to enter the data words you want recognized as mnemonics.

Above the VALUE field column there are two headings. The top heading indicates the source of data that is to be disassembled by the table. In the default configuration (i.e., with no changes made to the Table Definition sub-menu), the data source is the channel group associated with the table currently displayed. The second heading indicates the radix of the data values.

The radix of the channel group in the Channel Specification menu determines the radix of the data in the Table Entry sub-menu. Radix changes in the Channel Specification Sub-menu are automatically reflected in the Table Entry sub-menu.

You may enter mnemonics using one radix, then later change the radix in the Channel Specification menu for acquisition and disassembly.

You may change the number of VALUE field bits, but only before entries have been made in a table. To do so you must add or delete group channel inputs in the Channel Specification menu. In default, all bits in the VALUE field are in the don't care state.

During disassembly, tables are searched sequentially starting at sequence 0. If data received by the table matches the data in the field, then the sequence performs its part of the disassembly. If the data received by the table does not match the data specified, the table looks at the next sequence for a match. If two entries can both accept the same bit pattern, the entry with the lowest sequence number performs the disassembly.

Use the data entry keys to enter the desired value. Don't care values are entered by using the DON'T CARE key. The DAS enters the value at the cursor location, then moves the cursor one space to the right.

(5) DISPLAY Field

Once you have specified data values in the VALUE field, enter that value's corresponding mnemonic label in the DISPLAY field. In default, 10 characters are provided for State Table display of the mnemonic. Extra characters will be truncated from the right. (You can change the number of available display characters in the Display Setup sub-menu. Refer to the Display Setup Sub-Menu portion of this section for details.)

When entering the mnemonic, you may use the data entry characters (A-Z and 0-9), and the special symbols (available by pressing the pattern generator keys and SHIFT simultaneously). You can also use any of the other DAS 9100 character set symbols (available by pressing the SHIFT and SELECT keys simultaneously).

NOTE

Procedures for accessing the DAS Character Set are provided in Appendix B of this addendum.

To enter a mnemonic:

1. Move the screen cursor to the DISPLAY field.
2. Use the data entry keys, the special symbol keys (SHIFT/pattern generator keys), or the SHIFT/SELECT function to enter the desired mnemonic.

To remove a mnemonic:

Place the cursor anywhere in the DISPLAY field and press the DON'T CARE key.

DISPLAY SETUP SUB-MENU

To enter the Display Setup sub-menu, press the DEFINE MNEMONICS menu key and select the Display Setup mode (by pressing the SELECT key while the cursor is in the MODE field).

The Display Setup sub-menu allows you to format the display of data and mnemonics in the State Table. It lets you turn State Table display of data and mnemonics on or off, and lets you specify the character width allowed for each column of mnemonics on the State Table.

Figure 7-4 shows the relationship of the Display Setup sub-menu to the State Table menu.

NOTE

Certain details of the Display Setup sub-menu s use will change when the Define Mnemonics menu is in non-default configuration (i.e., when changes have been made in the Table Definition sub-menu). For more information, refer to the Disassembly Guidelines at the end of this section.

Figure 7-5 shows the Display Setup sub-menu and its default fields and values. Refer to this figure while reading the following paragraphs. The numbers are intended as a visual reference, and do not imply sequence of use.

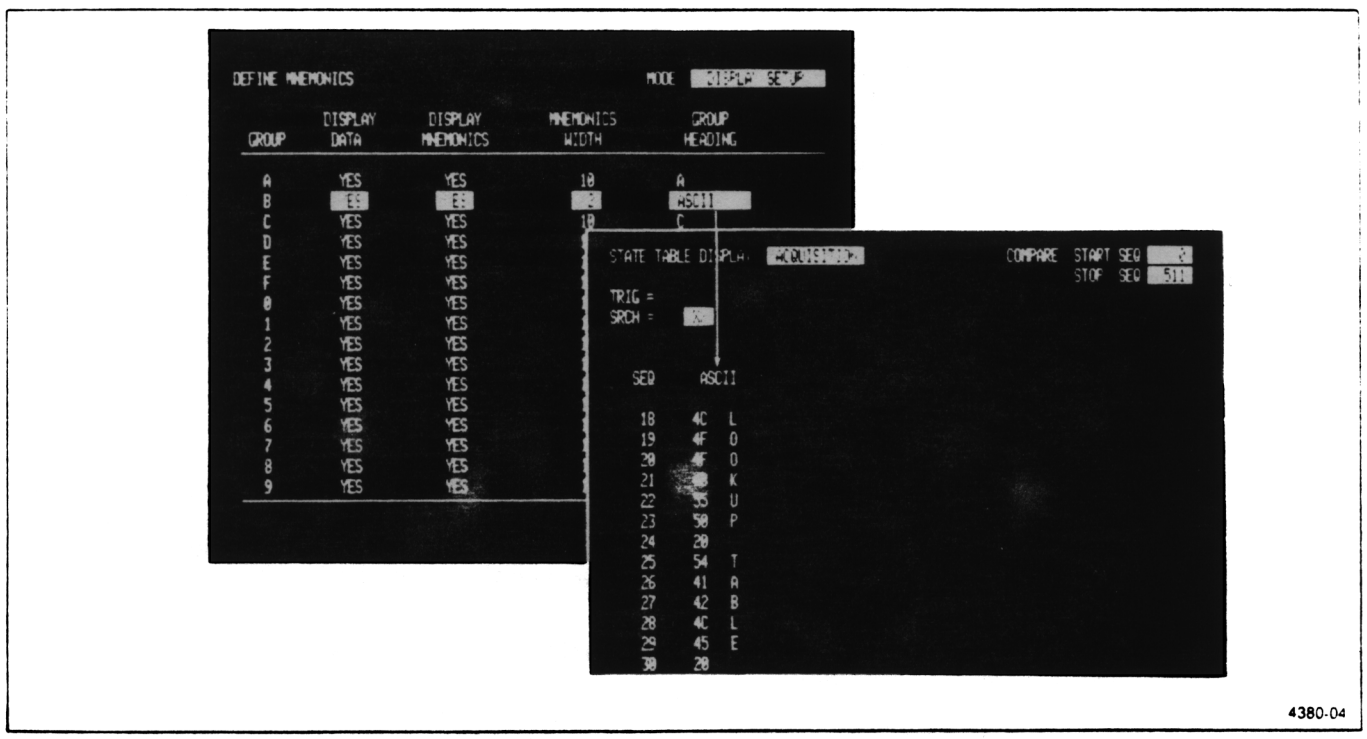
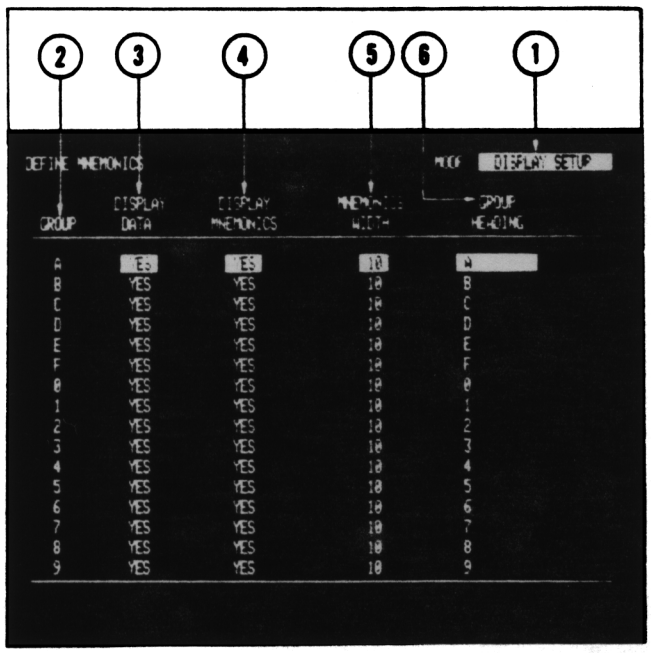


Figure 7-4. Relationship of the Display Setup sub-menu to the State Table Display. The Display Setup sub-menu formats the display of data and mnemonics in the State Table display. In this example, the default group heading (B) was changed to ASCII, and the default mnemonic width (10) was changed to 2 (to center the group heading over the column). For details, see the discussion of Display Setup sub-menu fields.



4380-05

Figure 7-5. Display Setup sub-menu and its fields.

(1) MODE Field

The MODE field is used to select between the Display Setup and Table Entry sub-menus. To move between these two sub-menus, press the SELECT key.

(2) GROUP Column

Each line on the Display Setup sub-menu formats the State Table's display of mnemonics for one channel group. The GROUP column shows which channel group is formatted by each line. In default operation, the name of the channel group is the same as the name of the table (e.g., data from channel group A is disassembled using table A).

(3) DISPLAY DATA Field

This field allows you to choose whether or not to display a channel group's acquired data on the State Table menu. The default value of the DISPLAY DATA field is YES (the data will be displayed).

To set State Table display of data to YES or NO:

1. Move the screen cursor to the DISPLAY DATA field.
2. Press the SELECT key.

   The value in the field alternates between YES and NO.

When the DISPLAY DATA field for a channel group is set to NO, the DAS will not display that group's acquired data on the State Table menu. You can use this feature to provide more room for other data and mnemonics on the State Table menu.

(4) DISPLAY MNEMONICS Field

This field allows you to turn off the display of a channel group's mnemonics on the State Table menu. The default value of the DISPLAY MNEMONICS field is YES (the mnemonics will be displayed).

To set State Table display of mnemonics to YES or NO:

1. Move the screen cursor to the DISPLAY MNEMONICS field.
2. Press the SELECT key.

   The value in the field alternates between YES and NO.

When the DISPLAY MNEMONICS field for a channel group is set to NO, the DAS will not display mnemonics next to that group on the State Table menu. You can use this feature to provide more screen space for other data and mnemonics on the State Table menu.

(5) MNEMONICS WIDTH Field

This field lets you specify how many characters will be allotted for display of mnemonics next to a channel group on the State Table menu. The default value is 10 characters. You can specify from 1 to 64 characters by placing the cursor in this field and using the data entry keys.

To conserve display space on the State Table menu, specify only the number of characters necessary to display your longest mnemonic.

It is possible to enter a mnemonic in the Table Entry sub-menu's DISPLAY field that has more characters than specified in the MNEMONICS WIDTH field. In this case, the mnemonics will be truncated (from the right) when they appear on the State Table menu.

(6) GROUP HEADING Field

This field allows you to specify a title which will be displayed over the mnemonics for a group's data. In default, a group's title will be the same as the group's name (i.e., A-F or 0-9). If you are using the default menu configuration, there is no need to change these values. If you do wish to change the heading displayed over the mnemonics column, enter a new heading in this field. See Figure 7-4 for an example.

MICROPROCESSOR DISASSEMBLY

Overview

Microprocessor disassembly is set up through the Table Definition sub-menu. This sub-menu lets you build a nested table structure where one master table can control the incoming acquisition data, then pass relevant bits to appropriate sub-table levels. This allows bit-by-bit disassembly of every incoming microprocessor instruction.

Figure 7-6 provides a conceptual overview of how the nested table structure works. In this example, the table structure is set up for the 8085A 8-bit microprocessor. You can modify this structure to fit any other type of microprocessor disassembly.

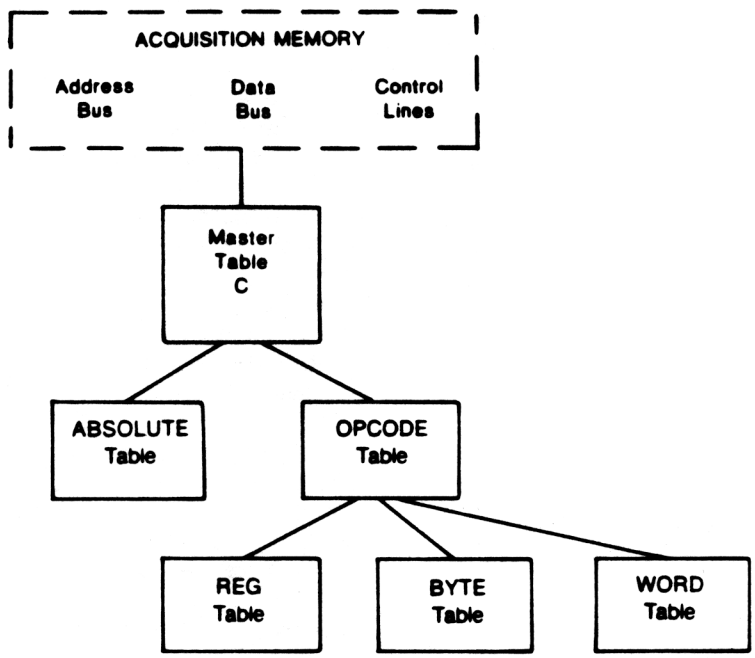


Table Definitions:

Group Table C	This master table receives the incoming data word, categorizes it by whether or not it is an instruction fetch, then calls tables and passes bits accordingly.
ABSOLUTE Table	This table contains mnemonic definitions for operations like memory read/writes, I/O read/writes, and interrupt acknowledges.
OPCODE Table	This table contains mnemonic definitions for the 8085A instruction set. This table can call and pass bits to the register table, or call either the byte or word table, depending on the instruction needs.
REG Table	This table contains names and decoding for the 8085A internal registers.
BYTE Table	This table does not receive passed bits. When called, it fetches the next data byte from memory and displays it in hexadecimal.
WORD Table	This table does not receive passed bits. When called, it fetches the next two data bytes from memory and displays them as a 16-bit word. (Note: This table calls and uses the functions of the Byte Table.)

4380-06

Figure 7-6. Example of a nested table structure using the 8085A microprocessor.

The nested table structure works in the following way. First, the incoming acquisition data is input to one of the group tables A-F or 0-9 (in this example, group table C). This table then serves as the master table for the structure.

Under the master table there is a series of user-defined nested tables. These nested tables are designed to break the data word down into smaller and smaller categories of recognizable bit patterns. Altogether, there can be up to 48 user-defined nested tables that can nest up to 16 levels deep.

In this example, the first level of nesting categorizes data by whether or not it is an instruction fetch. If the data is not an instruction fetch, it is passed from the master table to the absolute table which identifies such operations as memory and I/O reads and writes.

If the data is an opcode, it is passed from the master table to the opcode table. This table categorizes the data according to the microprocessor's instruction set. Depending on the complexity of the microprocessor, there may be several of these opcode tables, each supporting specific bit patterns.

Under the opcode table there are general-purpose support tables. These tables are designed to support often-used instruction parameters, such as source and destination operands.

Overall, the structure of the master table and its nested tables is highly flexible. The nested levels, table names, and table values can all be tailored to your specific microprocessor.

The following parts of this section describe how you use the sub-menus to set up a nested structure and its elements.

TABLE DEFINITION SUB-MENU

The Table Definition sub-menu is used to enter the framework for your nested table structure. In this sub-menu, you define the new nested tables and their names, specify data input to the tables, and determine which tables can call other tables. Once you have entered the framework for the table structure in this sub-menu, you then use the Table Entry sub-menu to implement the table values and actions.

NOTE

The table structure must be entered in the Table Definition sub-menu before you can enter values in the Table Entry sub-menu.

The Table Definition sub-menu is entered by pressing the SHIFT and DEFINE MNEMONICS keys simultaneously.

Figure 7-7 illustrates a typical display of the Table Definition sub-menu and its fields. In this example, the framework for the table structure in Figure 7-6 has been entered into the sub-menu.

The following paragraphs describe each of the sub-menu fields and show how they are used. Refer to the numbered callouts in Figure 7-7 when reading these field descriptions. The numbers are intended as a visual reference, and do not imply sequence of use.

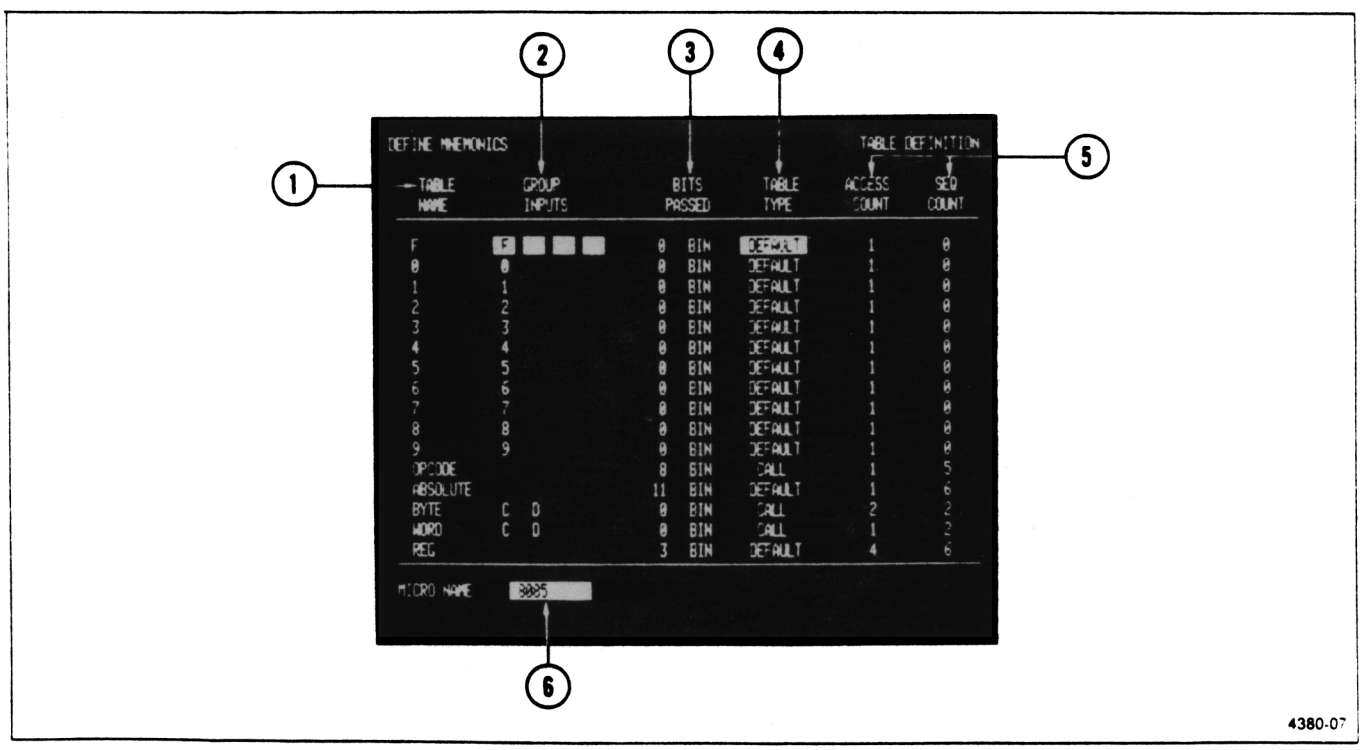


Figure 7-7. The Table Definition sub-menu and its fields.

(1) TABLE NAME Field

The TABLE NAME field is used to enter the tables for the nested table structure.

The TABLE NAME field always contains the group tables A-F and 0-9. These tables are always available and can serve as master tables in the nested structures. They can also serve as simple lookup tables as described in the front portion of this section.

In addition to the group tables, you can create up to 48 nested tables.

To create a nested table:

Use the ^ (scroll up) key to scroll to the end of the group tables. Then, in the blank field, use the data entry keys to enter the nested table's name. The name can be up to eight characters long.

To change the name of a nested table:

Move the screen cursor into the TABLE NAME field to be changed. Use the data entry keys to enter the new name over the old name. Any calls in the Table Entry sub-menu to the old table name will be changed to the new name.

The TABLE NAME field also provides a special function that lets you quickly access a table for display in the Table Entry sub-menu.

To display a table quickly in Table Entry:

Scroll the table you want to display to the top line of the screen, then press the DEFINE MNEMONICS key and enter the Table Entry sub-menu. The Table Entry sub-menu automatically displays the table that was located on the top line.

(2) GROUP INPUT Field

Use the GROUP INPUT field to control which tables have direct access to the acquired data. The master table and any nested tables that look at the next data word must have values specified in the GROUP INPUT field.

The following paragraphs describe how group inputs are used.

Setting Up The Master Table And Its Data Input. In default, each of the group tables receives data input from its corresponding channel group. When establishing a nested table structure, you need to change this default configuration so that one group table receives data from all channel groups associated with the program data (i.e., address, data, and control information). This group table then serves as the master table.

Before establishing the master group table and its channel group inputs, you must first consider how you have set up the channel group format in the Channel Specification menu. It is recommended that you set up the channel group format as follows:

Address Bus	—	Channel Group A
Data Bus	—	Channel Group D
Control Lines	—	Channel Group C

NOTE

While the above group format is only recommended, special features called System Calls work on the assumption that you have grouped the channels in this manner. For more information, refer to System Calls later in this section.

Once you have determined the channel group format, you then select the master group table by specifying these channel groups as inputs to the table.

To select the master group table and its data input:

Move the screen cursor to the group input fields of the group table you want to serve as the master table. Then, using the data entry keys, enter the channel group inputs. Up to four channel groups can be specified.

Any of the group tables (A-F and 0-9) can be used for the master table. However, to ensure proper disassembly, you should select a group table that corresponds to one of the channel groups acquiring data. It is recommended that you select group table C for compatibility with the system calls.

For more information regarding the relationship of the master table to its group inputs, refer to Disassembly Guidelines given later in this section.

Setting Up A Nested Table To Access The Next Data Word. A nested table can receive data in two ways: group input and passed bits. The following paragraphs describe the reasons for using group input to a nested table. The reasons for using passed bits are described under the PASSED BITS Field.

Many microprocessor instructions require more than one byte of sequential data. For example, an 8085A's MOVI (Move Immediate) instruction requires the use of the next eight bits of data. To get these next eight bits, you must establish a nested table that, when called by another table, goes to memory and accesses the next sequential data word. You establish this nested table by giving it group inputs.

To access the next data word:

Move the screen cursor to the GROUP INPUT fields of the nested table you want to access the next data. Then, using the data entry keys, enter the channel groups which contain the data you want the table to access. To

ensure proper disassembly, one of these group inputs should be the channel group which corresponds to your master table.

NOTE

The number of channels that are assigned to a table cannot change once values are entered in that table. Plan carefully when defining your table structure.

(3) BITS PASSED Field

A nested table can receive data in two ways: group input and passed bits. The following paragraphs describe the reasons for using passed bits. The reasons for using group input are described under the GROUP INPUT field.

NOTE

This field only applies to nested tables. Tables A-F and 0-9 cannot receive passed bits.

The purpose of a nested table structure is to take a data word and send it through increasingly detailed disassembly. To do this, the various nested tables of the structure must be able to receive pieces of the data word from other tables. The amount and type of data the table needs to receive is dependent on the table's disassembly function.

You specify whether a nested table can receive data bits in the PASSED BITS field. Any individual nested table can receive up to 32 bits.

NOTE

The number of bits passed to a table cannot change once values are entered in that table.

To specify the number of bits passed:

Move the screen cursor to the bit field of the nested table you want to receive data. Then, use the data entry keys to specify from 1 to 32 bits.

In addition to specifying the number of bits, you use the BITS PASSED field to specify the radix of the bits as they appear in the Table Entry sub-menu.

To specify the radix of bits passed:

Move the cursor to the radix field, then press the SELECT key. The field displays the optional radix values in this order:

[BIN]
[HEX]
[OCT]

NOTE

The selected radix does not affect the number in the BITS PASSED field. The number in this field always refers to binary bits.

(4) TABLE TYPE Field

The TABLE TYPE field is used to specify whether or not a table can call other tables.

In default, all tables are a DEFAULT type. They operate as simple lookup tables and can be called by other tables, but they cannot call other tables.

You can change any of the tables to a CALL type. This means that the table can call other tables. In a nested structure, the master table, and any nested tables that rely on other nested tables, must be a CALL type.

NOTE

A table must also be a CALL type if it accesses any system calls. For more information, refer to the System Calls description later in this section.

To change a table to a CALL type:

Move the screen cursor to the table's TABLE TYPE field, then press the SELECT key. The field displays optional values in this order:

[DEFAULT]
[CALL]

(5) ACCESS COUNT and SEQ COUNT Columns

These two columns keep track of the various tables once you start entering values in the Table Entry sub-menu.

The SEQ COUNT column keeps track of how many sequence values each table contains. In default, the number of sequences for each table is zero, which means the tables are empty. This number will increment when you start entering table sequence values. To decrement the sequence count use the DEL LINE key to remove the entry from the table.

The ACCESS COUNT column keeps track of how many times a table is accessed or called by another source.

In default, the access count value for each group table A-F and 0-9 is one. These tables will always have an access count of at least one because they are automatically accessed by their corresponding group column on the State Table.

Each nested table starts out with an access count of zero. This count value will increment each time you call the table from another table.

NOTE

The access count for nested tables must be zero before you can change the table's GROUP INPUT or BITS PASSED fields.

(6) MICRO NAME Field

This field allows you to specify the name of the processor for which your mnemonic disassembly is designed. The name entered in the field may then be displayed in the State Table and Trigger Specification menus.

The name entered in the MICRO NAME field can be displayed in the column headings in the State Table menu. Before the name will be displayed, however, the following conditions must be met:

- The name only appears over the group A, group C, and group D columns of the State Table.
- The name only appears over a group column in the State Table when the DISPLAY MNEMONICS field for that group is set to YES, in the Display Setup sub-menu.
- The display width for the group column in the State Table display must exceed the length of the name in the MICRO NAME field by at least five characters. The display width is the sum of the mnemonic width (as defined in the Display Setup sub-menu) and the data display width.

DEFINE MNEMONICS — TABLE DEFINITION

TABLE NAME	GROUP INPUTS	BITS PASSED	TABLE TYPE	ACCESS COUNT	SEQ COUNT
A	-	0 BIN	DEFAULT	1	0
B	B	0 BIN	DEFAULT	1	0
C	A C D	0 BIN	CALL	1	0
D	D	0 BIN	DEFAULT	1	0
E	E	0 BIN	DEFAULT	1	0
F	F	0 BIN	DEFAULT	1	0
0	0	0 BIN	DEFAULT	1	0
1	1	0 BIN	DEFAULT	1	0
2	2	0 BIN	DEFAULT	1	0
3	3	0 BIN	DEFAULT	1	0
4	4	0 BIN	DEFAULT	1	0
5	5	0 BIN	DEFAULT	1	0
6	6	0 BIN	DEFAULT	1	0
7	7	0 BIN	DEFAULT	1	0
8	8	0 BIN	DEFAULT	1	0
9	9	0 BIN	DEFAULT	1	0
OPCODE		8 BIN	CALL	0	0
ABSOLUTE		11 BIN	DEFAULT	0	0
BYTE	C D	0 BIN	CALL	0	0
WORD	C D	0 BIN	CALL	0	0
REG		3 BIN	DEFAULT	0	0

MICRO NAME 8085

4380-08

Figure 7-8. Table Definition values for disassembling an 8085A microprocessor. The ACCESS COUNT and SEQ COUNT fields are still at their default values because no tables for disassembling the 8085A have yet been entered. Note that tables A, B, D-F, and 0-9 are left as they were in the default situation because they are not used for disassembly.

The name entered in the MICRO NAME field is also displayed in the Trigger Specification menu whenever a PMA 100 Personality Module Adapter is connected to the DAS.

To enter a micro name:

Move the cursor to the bottom of the DAS display. Then, in the blank field, use the data entry keys to enter the processor's name. The name can be up to nine characters long.

To delete a micro name:

Place the cursor in the MICRO NAME field, then press the DON'T CARE key. This blanks the entire MICRO NAME field.

Defining a Mnemonic Structure

As mentioned previously, the table structure for any disassembly tables must be entered into the Table Definition sub-menu before entering values into the Table Entry sub-menu.

Taking the previously developed example of an 8085A disassembly structure (refer to Figure 7-6), the values shown in Figure 7-8 would be entered in the Table Definition sub-menu to disassemble an 8085A microprocessor. Refer to Figure 7-8 while reading the following text.

Table C (Figure 7-10) is the master table in this example. It receives all address (group A), data (group D), and control lines (group C) as specified in the GROUP INPUTS field. The TABLE TYPE field is set to CALL so the master table can call the nested tables.

Table ABSOLUTE has data and control bits passed to it. This table disassembles operations like memory reads and writes, I/O reads and writes, and interrupt requests. Notice that the ABSOLUTE table is a DEFAULT table since it does not need to call other tables.

Table OPCODE (Figure 7-11) also has data and control bits passed to it. But, unlike table ABSOLUTE, OPCODE is a CALL table since it calls other tables to complete disassembly of most instructions.

Table BYTE (Figure 7-12) does not have any bits passed to it. BYTE's function is to look at the next acquired word (so it has group inputs C and D) to decode operands of the current machine instruction.

Most other nested tables, like WORD and REG, are similar to one of the three nested tables just described.

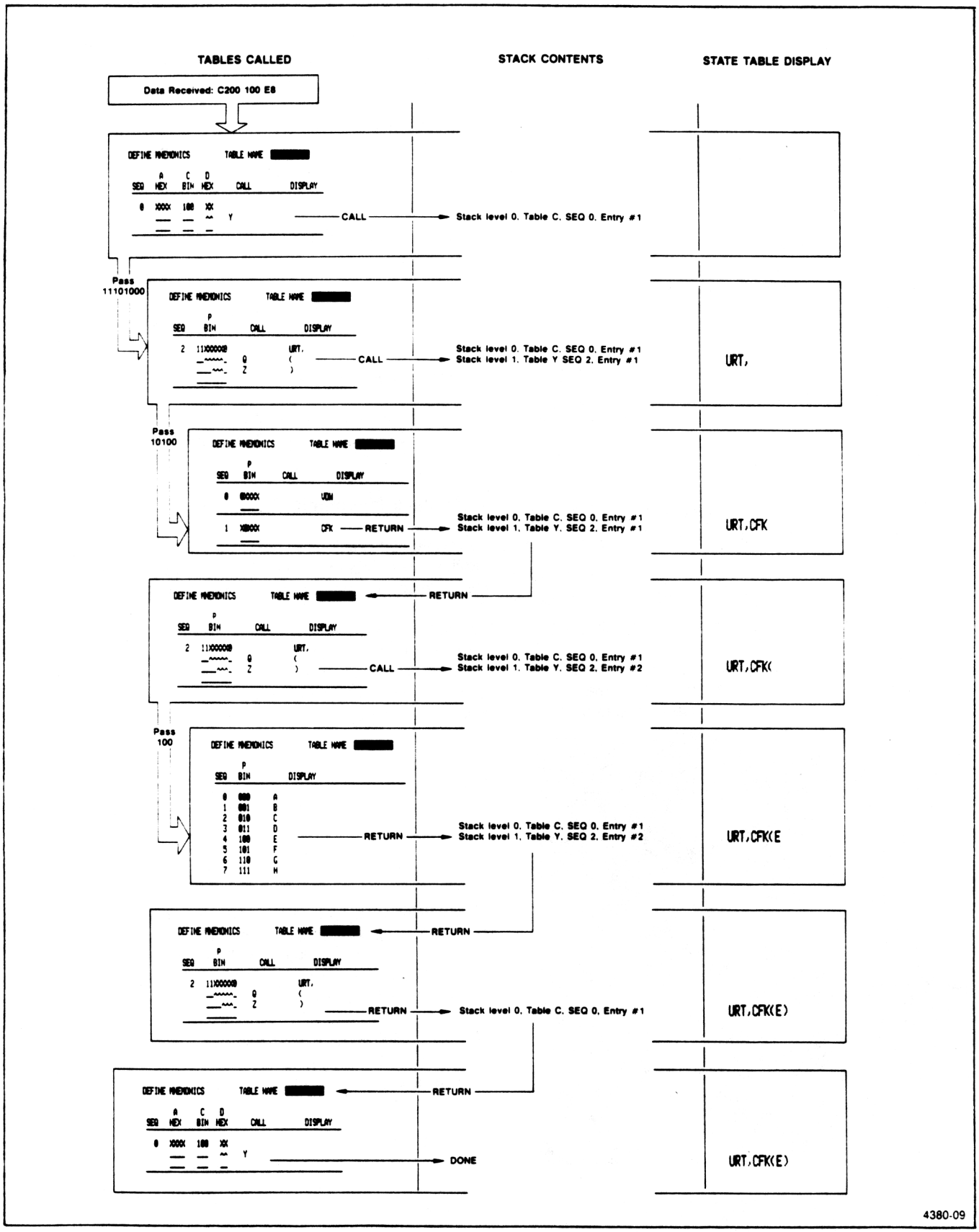


Figure 7-9. Overview of how table calls operate. This figure shows how using call tables for disassembly is like a program which calls and returns from subroutines. The locations that must be returned to are saved in a push-down stack. Each of the called tables may add a portion to the displayed mnemonic.

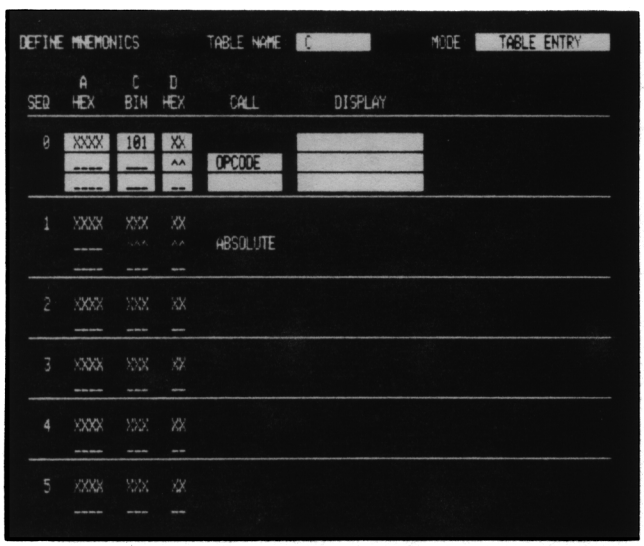
USING CALL TABLES IN THE TABLE ENTRY SUB-MENU

Call tables occur in the Table Entry sub-menu of the Define Mnemonics menu when CALL is selected in the TABLE TYPE field of the Table Definition sub-menu. Call tables can call disassembly procedures that are set up in other tables. By spreading mnemonic disassembly through several tables that act as subroutines, disassembly of complex microprocessors is possible without immense lookup tables. Tables that function as subroutines are called using the CALL field.

When a table is called by another table, disassembly moves from the current table to the beginning of the called table. The called table may also call other tables (tables may be called up to 16 deep). When any called table reaches the end of a sequence, disassembly returns to the table that called the completed table. Figure 7-9 shows how the call structure of disassembly works.

It may be helpful to consider called tables as a set of subroutines in a program. There is a main program sequence, the master table, which calls subroutines. Each subroutine may then call other subroutines. When a subroutine finishes, the program returns to the next step in the subroutine that initiated the call. A stack keeps track of each subroutine call, so the disassembly can return to the correct location in the table that called the subroutine.

The following discussion of call tables describes how to use the CALL field, how to pass bits to called tables, and how to access the next data word. The discussion focuses on disassembly of the 8085A microprocessor. However, other processors can be disassembled using the techniques shown here.



4380-10

Figure 7-10. Table C (master table) for 8085A disassembly. This table seperates opcode fetches from other types of machine cycles (like memory reads and writes and I/O reads and writes).

These paragraphs describe each of the sub-menu fields and show how they are used. Figure 7-10 illustrates a typical display of a call table. Refer to the numbered callouts in Figure 7-10 when reading the field descriptions. The numbers are intended as a visual reference, and do not imply sequence of use.

(1) CALL Field

The CALL field is used to specify the table or system call that the disassembly routine goes to next.

There are two table calls in Figure 7-10, one in SEQ 0 to table OPCODE and one in SEQ 1 to table ABSOLUTE.

To call a table, enter that table's name in the appropriate CALL field. A return is performed when disassembly reaches the end of a sequence in the called table.

NOTE

Any table name entered in the CALL field must match one of the table names already set up in the Table Definition sub-menu. The cursor will not leave a CALL field containing an undefined table name.

To enter a CALL value:

1. Move the screen cursor to the CALL field.

2. You may use the data entry keys, A-Z and 0-9, and the punctuation keys to enter the desired name in the field. Names entered may be a maximum of eight characters long. For example, ABSOLUTE.

   [ABSOLUTE]

3. The DAS displays the entered table name in the field. Any mnemonic disassembly that reaches this point performs the disassembly in table ABSOLUTE.

If you look in the Table Definition sub-menu (by pressing the SHIFT/DEFINE MNEMONICS keys), you will see that the ACCESS COUNT value for table ABSOLUTE has incremented by one. The ACCESS COUNT of a table increments every time that table is called by another table.

To delete a CALL value:

Move the cursor to the call value to be removed and press DON'T CARE. The field will blank.

(2) VALUE Field

The VALUE field of the Table Entry sub-menu is used to enter the data which will be recognized as mnemonics. The field is also used to pass bits from the current table to a called table.

VALUE Field Column Headings. Before entering anything in the VALUE field, note the labels at the top of each of the columns in the VALUE field. Each column head corresponds to inputs to the table as specified in the Table Definition sub-menu. In Figure 7-10 these inputs are group A, group C, and group D. If bits were passed to the table, there would also be a column labeled P (for pass).

Specifying Values. You may specify a data value in the VALUE field on the first line of any sequence. If data received by the table matches the value in the field, then the sequence performs its part of the disassembly. If the data received by the table does not match the value specified, the table looks at the next sequence for a match. So if two entries can both accept the same bit pattern, the entry with the lowest sequence number performs the disassembly.

This hierarchical design can reduce the number of entries in a table. Note that in Figure 7-10 sequence 0 requires a specific bit pattern to call table OPCODE. This bit pattern corresponds to an instruction fetch by an 8085A microprocessor. Sequence 1 accepts any bit pattern at all, but it only receives bit patterns that are not instruction fetches because sequence 0 intercepts all instruction fetches.

To specify or change a value:

1. Move the screen cursor to the topmost value field of a sequence. This field contains all Xs (don't cares) in default.

2. Using the data entry keys, enter the desired value in the radix indicated at the top of the column. X (don't care) is specified with the DON'T CARE key.

To delete an entire sequence:

1. Move the screen cursor into the topmost value field of the sequence you want to delete.

2. Press the DEL LINE key. The sequence will disappear and any sequences below the deleted sequence will move up.

Passing Bits. Whenever a table is called that has bits passed to it, the bits to be passed must be specified in the value field. The bits passed to the table are indicated by the ^ (SHIFT/H-ALT) character on the DAS screen.

NOTE

When passing bits to a table, you must pass exactly the number of bits expected by the called table. The cursor will not leave the field if you pass the wrong number of bits.

Data passed from one table to the next need not match radices, but the number of binary bits sent must equal the number of binary bits expected by the called table. For example, passing a hexadecimal digit passes four bits, passing an octal digit passes three bits, and passing a binary digit passes one bit.

To indicate the bits passed to a table:

Move the screen cursor into the value field to the left of the table call.

```
   HEX        BIN        HEX
[_ _ _ _] [ _ _ _ ] [  _ _  ]  [ ABSOLUTE ]
```

Press the SHIFT/HALT keys (^) at every location that you want to pass bits.

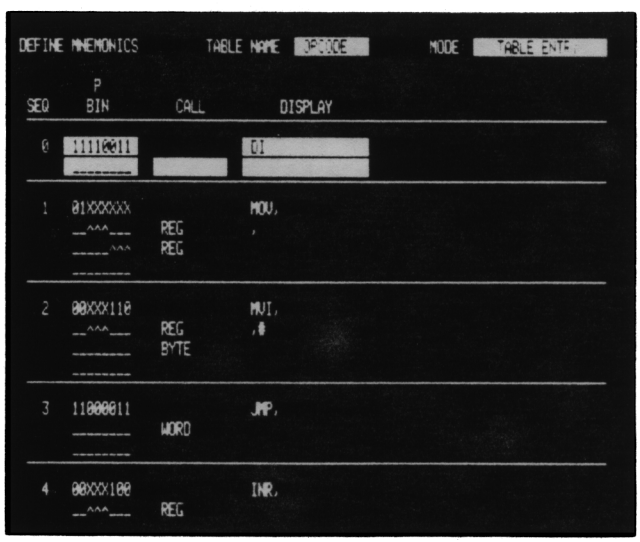
```
   HEX        BIN        HEX
[_ _ _ _] [ ^ ^ ^ ] [  ^ ^  ]  [ ABSOLUTE ]
```

The ^ indicates the bits to be passed to the next table. In this example, table ABSOLUTE will receive 11 bits.

Passed bits may be removed by moving the cursor over the ^ to be removed and pressing DON'T CARE.

(3) DISPLAY Field

For an example of how the DISPLAY field may be used in call tables, refer to Figure 7-11. This figure shows how table OPCODE uses the DISPLAY field in conjunction with the CALL field.



4380-11

Figure 7-11. Table OPCODE for 8085A disassembly. This table is called by table C (the master table) to translate 8085A opcodes into mnemonics. Note that this table, which acts as a subroutine for table C, also calls other tables.

When using the DISPLAY fields in call tables, it is useful to think of disassembly moving from left to right, then top to bottom. Disassembly happens in the following sequence:

1. The table receives data and locates a sequence that matches this data. The data may include both passed bits and group data from the State Table.

2. The Define Mnemonics menu writes the characters from the first DISPLAY field of the table sequence on to the State Table display. For example, if 01000111 were passed to table OPCODE in Figure 7-11, the disassembly would start at table sequence 1 and "MOV," would be displayed on the screen.

3. Disassembly moves down to the next line of the table sequence. If there is a name in the CALL field, the table is called. Disassembly proceeds to the called table, in this case table REG.

4. When table REG is finished, disassembly returns to the current table and whatever is in the DISPLAY field next to the table call is displayed on the State Table (a comma).

5. Disassembly continues in this order until all table calls and displays have been performed. In this example the State Table would display "MOV,B,A". The B and A would be decoded by table REG.

NOTE

Display entries are placed adjacent to one another without spaces between them. To place spaces between display entries, system calls must be used. System calls are described later, under the heading System Calls.

To enter characters in the DISPLAY field:

Move the screen cursor into the field. Enter characters with the data entry keys 0-9 and A-Z, and the punctuation keys.

To delete a display entry:

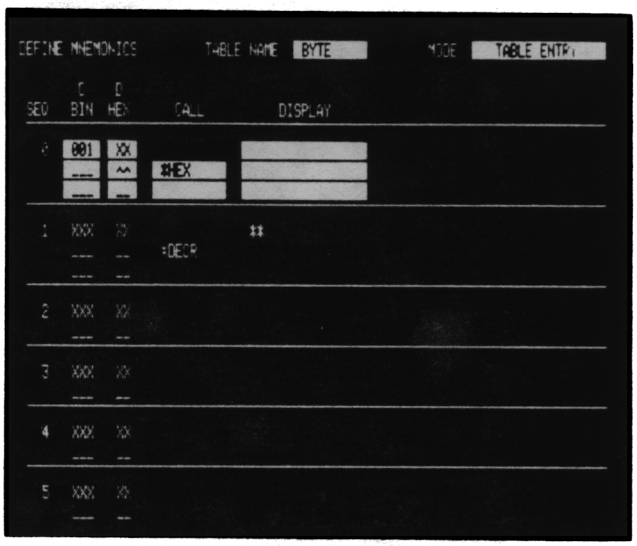
Move the screen cursor into the display field to be blanked. Press the DON'T CARE key. The field will be blanked.

Accessing the Next Word

Many microprocessor instructions are fetched by the processor over more than one machine cycle. In the 8085A microprocessor there are instructions with one or two operands that must be fetched before the instruction cycle is complete. The Z80 microprocessor has some opcodes that occupy two successive memory locations as well as having operands.

To disassemble instructions that occupy several machine cycles, the Define Mnemonics menu uses call tables to work on several consecutive sequences of acquired data.

An Example with the 8085A. One example of an instruction that takes two machine cycles is the MVI (move immediate) instruction of the 8085A microprocessor. Table OPCODE, shown in Figure 7-11, decodes a move immediate instruction in table sequence 2. Sequence 2 displays the next acquired word by calling table BYTE. Table BYTE is shown in Figure 7-12. Notice that no bits are passed to table BYTE.



4380-12

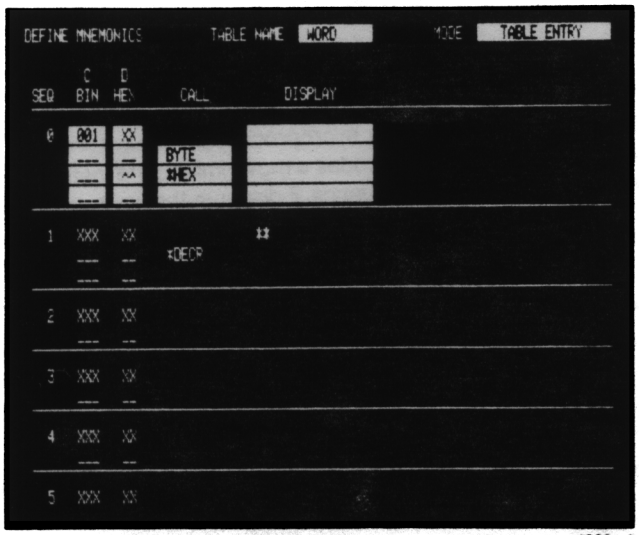
Figure 7-12. Table BYTE for 8085A disassembly. This table is called by table OPCODE to display the next byte of data. Table BYTE has group C and D inputs, but receives no passed bits. The group inputs send the next acquired data to the table.

*HEX is a system call that displays the bits passed to it in hexadecimal radix. System calls are described in detail later, under the heading System Calls.

By accessing successive words, multiple byte instructions can be decoded. Table BYTE checks the control lines to make sure the byte is part of the instruction, then the acquired data is displayed in hexadecimal radix. Disassembly then returns to table OPCODE.

Table WORD (shown in Figure 7-13) provides an example showing which word is disassembled by each table. Table WORD is called by OPCODE to display the next two trailing data bytes as one word. The second byte must be displayed first since the second byte is the high-order half of the word and the first byte is the low-order half of the word.

Tables, other than the master table, only receive new data when they are called. Returning from a table does not bring new data into a table, even if the table has group inputs. After returning from a called table, the calling table's disassembly continues as though uninterrupted.



4380-13

Figure 7-13. Table WORD for 8085A disassembly. Table WORD is called by OPCODE (shown in Figure 7-11) to display the next two acquired bytes from the microprocessor data bus as one 16-bit word. The first acquired word contains the low-order byte, and the second acquired word contains the high-order byte.

Table WORD receives word 2 (the first word after OPCODE's word - word 1), but does not display it yet. WORD calls BYTE, which displays word 3 (the second word after OPCODE's word), and returns to WORD. Then WORD displays the data it received (word 2) and returns to table OPCODE.

An Example with the 6809. On the other hand, consider disassembling a two-byte word for a 6809 microprocessor. With the 6809 (in contrast to the 8085), the first byte is high-order, and the second byte is low-order. So the first byte should be displayed first and the second byte displayed second. With this high-low byte order, table WORD is no longer needed. It can be replaced by calling table BYTE twice as shown in Figure 7-14.

Figure 7-14 shows that once a word has been used, it is not used again by any other table in the structure. Calling BYTE twice in Figure 7-14 displays two consecutive words, not the same word twice. (Some system calls can be used to override this, allowing words to be dissassembled more than once. For more details see the System Calls description.)

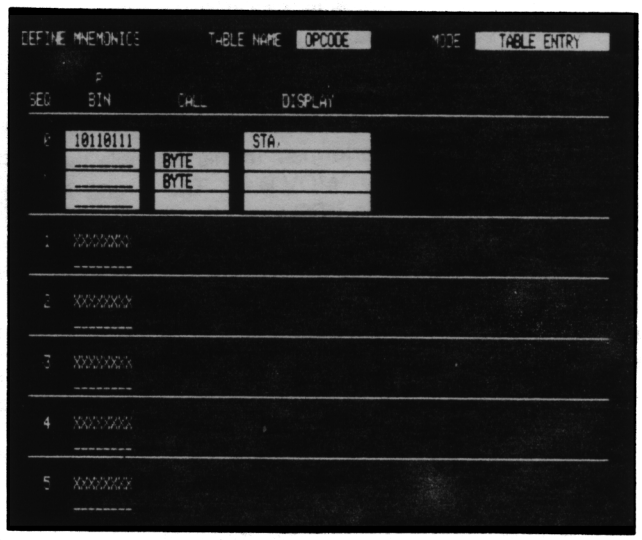
Disassembling a Series of Machine Instructions

When an opcode is entirely disassembled, the Define Mnemonics menu starts disassembly of the next opcode by calling the master table. The following rules indicate when the master table is called.

1. The word at the top of the State Table screen (the one with the lowest sequence number) is the first word sent to the master table. The Define Mnemonics menu disassembles only the data currently displayed on the State Table. New data is disassembled when it is moved into the display area of the State Table menu, either by scrolling or changing sequence numbers.

2. When a machine instruction is completely disassembled (there are no more call or display instructions for that instruction) the next acquired data that has not been disassembled is sent to the master table. This continues until the State Table display is filled or there is no more acquired data.

3. The master table determines the next data to use by looking at the last-used data in its own group (e.g., table C checks group C acquired data). The master table assumes that the last data used in its group corresponds to the last data used in all of its accessed groups.



4380-14

Figure 7-14. Table OPCODE for disassembling an extended address mode STA (store accumulator in memory) of a 6809 microprocessor. The hexadecimal machine code B7 is detected, which indicates an extended address STA instruction. Table BYTE is then called twice in a row. The first call to BYTE displays the first (high) byte after the STA opcode. The second call to BYTE displays the next (low) byte, which is the second byte after the STA opcode.

SYSTEM CALLS

Define Mnemonics System Calls are special commands which facilitate mnemonic disassembly. They give you added capability to control the disassembly and display of acquired data. You can use them in any call table.

To enter a system call:

You enter a system call in the same way you enter a table call: by entering the name of the system call in a Table Entry sub-menu CALL field. (System calls are essentially reserved table names, and are always preceded by an asterisk.) Some of the system calls require you to pass bits. You do this in the same manner as you would pass bits to a table.

During disassembly of a data word, most system calls behave in the same nested fashion as table calls. When the DAS has executed a system call, it returns to the next line of the call table's disassembly sequence.

The following paragraphs describe each of the system calls and their use.

System Calls for Controlling Mnemonic Display

These system calls control how disassembled data is displayed on the State Table.

***BLANK.** This system call displays one blank character. You can use it to separate different groups of displayed data (for example, opcodes from operands).

***TAB.** This system call also allows you to separate displayed data by advancing display of the next item to the next tab setting. Tabs are preset to character positions 1, 9, 17,...57.

***HEX.** This system call displays the hexadecimal value of the bits passed to it. It does not suppress leading zeroes.

***OCT.** This system call works in the same manner as *HEX but displays an octal value.

***BIN.** This system call works in the same manner as *HEX but displays a binary value.

***HEXS.** This system call displays the value of the bits passed to it in signed hexadecimal.

NOTE

**HEXS only works on 8 or 16 bits. If you pass the wrong number of bits to *HEXS, negative word values can cause the wrong number to be displayed.*

*HEXS assumes the bits passed to it form a two's complement number. It displays a plus sign if the most significant bit is a 0, or a minus sign if the most significant bit is a 1.

The example in Figure 7-15 shows how some of the system calls for controlling mnemonic display might be used.

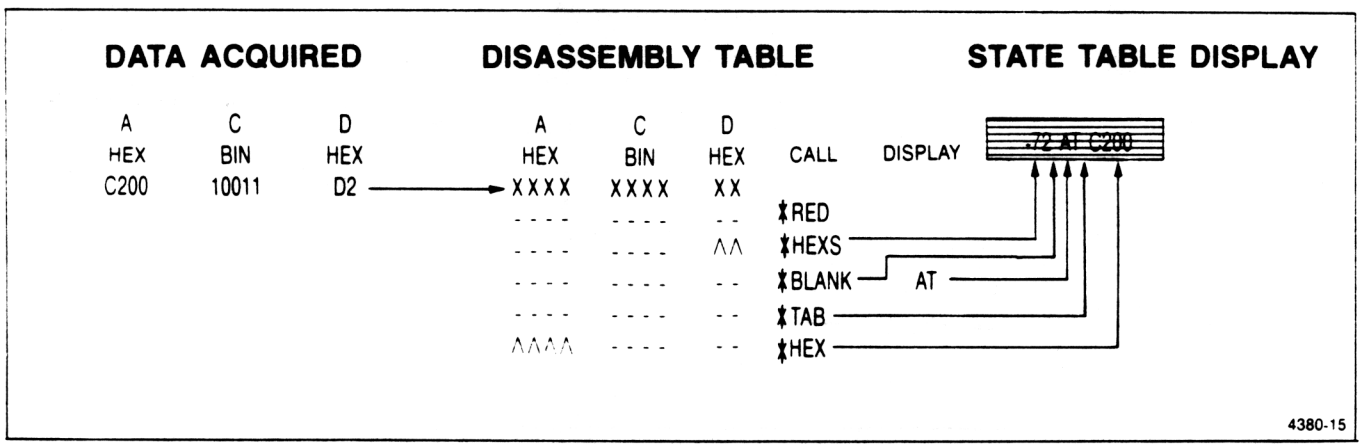


Figure 7-15. Using *RED, *HEXS, *BLANK, *TAB, and *HEX. This example demonstrates how the display system calls can organize the display of data. The highlighted display line is caused by the *RED system call.

***RED (highlight).** This system call sets the display to the color red, or, in the case of a monochrome display, causes a highlighted display.

***GREEN.** This system call works in the same way as *RED but causes a green display. *GREEN has no effect on a monochrome DAS display.

***YELLOW.** This system call works in the same way as *RED but causes a yellow display. (Note that yellow is the default color for State Table display of disassembled data.) *YELLOW has no effect on a monochrome DAS display.

All of the color calls affect the entire line of the group displayed on the State Table, not just a single item. If more than one color is called during disassembly of a machine instruction, the last color specified is the one displayed.

System Calls for Relative Addressing

The four *RLADD system calls perform relative address calculations. For example, when disassembling a relative jump instruction you can use a *RLADD system call to display the actual address jumped to, rather than displaying the offset. The *RLADD system calls operate on addresses up to 32 bits long.

NOTE

*All of the *RLADD system calls assume that the microprocessor address lines are acquired through group A (in the Channel Specification menu). Values are displayed in the radix of group A.*

Table A can receive a maximum of 32 bits from combined inputs when relative adds are used. All other tables can receive 138 bits.

All of the *RLADD system calls use the value in group A as their base and then add or subtract an offset from this group A value. Do not pass bits from group A to the *RLADD system call; the *RLADD system call automatically gets the value from group A for you. The group A data used is the value in group A that matches the most recently looked at group C data. Figure 7-16 illustrates which group A data is used by *RLADD calls.

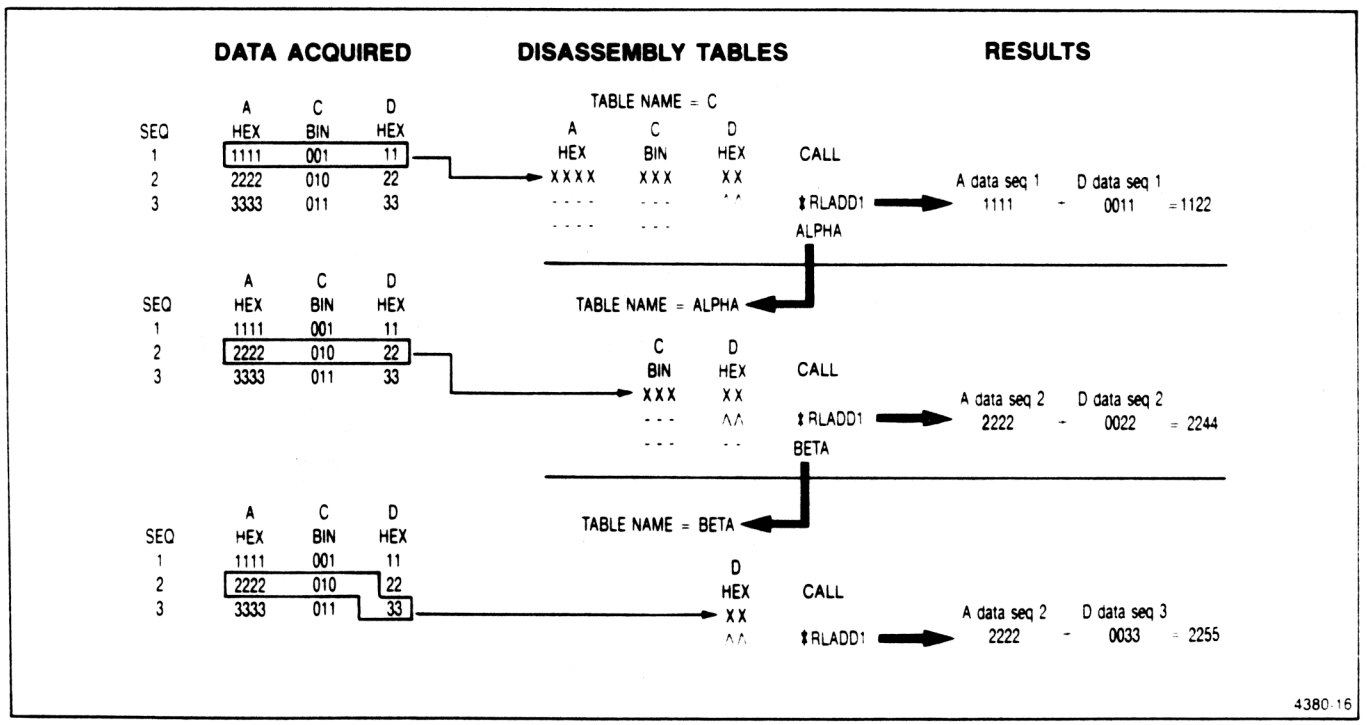


Figure 7-16. The *RLADD system calls use group A data from the data sequence most recently looked at by group C. Table BETA provides an unexpected result because the group A and group D data read by table BETA do not come from the same data sequence. Because table BETA did not access group C, the last data sequence looked at by group C was data sequence 2.

The data (offset) you pass to a *RLADD call is treated as a two's complement number. The most significant bit is treated as a sign bit. Each of the *RLADD calls display the resulting number in whatever radix group A is set to. Details specific to each of the *RLADD system calls are provided in the following paragraphs.

NOTE

*RLADD system calls only operate properly if you pass 8, 11, or 16 bits to them.

***RLADD1.** (Group A + offset). This system call adds the value in group A to the bits you pass to it.

***RLADD2.** (Group A + 1 + offset). The *RLADD2 system call adds 1 to group A and the bits you pass to it. This method of relative addressing is typically used by 8-bit processors.

***RLADD3.** (Group A + 2 + offset). *RLADD3 adds 2 to group A and the bits you pass to it. Sixteen-bit processors with byte addressing typically use this method of relative addressing.

***RLADD4.** [Group A + 2 + (2 x offset)]. *RLADD4 adds group A plus two plus two times the bits you pass to it. This method of relative addressing is used by Z8000 and DEC PDP 11 processors.

The example in Figure 7-17 shows how *RLADD1 might be used.

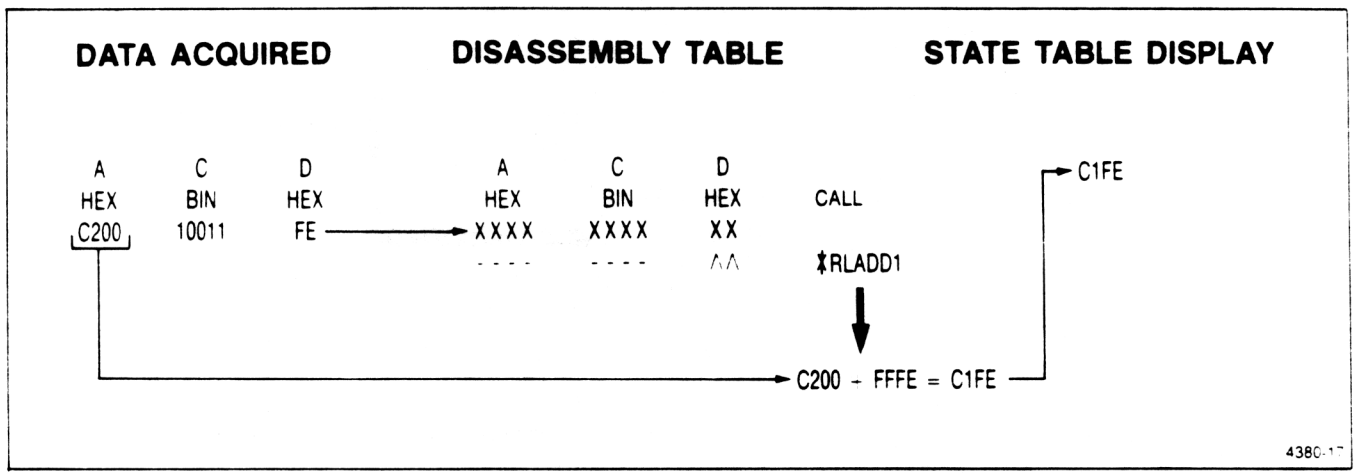


Figure 7-17. Using *RLADD1 to calculate a relative address. This table could be called by another table that discovered a relative branch or jump instruction. Note that the group A bits are not passed to *RLADD1; the system call automatically gets the group A data.

Error-Handling System Calls

***ERROR.** This system call erases any disassembly which has occurred so far on the current machine instruction, then calls the table named ERROR*.

You must create the table named ERROR*. It allows you to define what information about the error you want to display. ERROR* may only have group inputs. The usual inputs are the address and data groups. The control line group may also be an input to show the bus cycle type.

ERROR* operates on the data sent to the master table for the current machine instruction. So even though the table has group inputs, it does not access the next data. In fact, it may move backwards through the disassembled data to arrive at the point where the master table was last called.

Once table ERROR* is finished, the next data in acquisition memory is automatically sent to the master table to start disassembling the next machine cycle.

***EXIT.** The *EXIT system call displays all of the disassembly performed up to the *EXIT, then terminates disassembly of the current data word. The next word is then disassembled starting from the master table, and displayed on the next line of the State Table.

***DECR.** The *DECR system call backs up the data used during disassembly by one acquisition data sequence. After a *DECR, the next table call (master table or nested table) that accesses group data directly (not passed bits) receives the same data word that the table containing *DECR was looking at.

The effect of the *DECR system call continues even when disassembly is restarted and the master table is called for the next instruction. If several *DECRs are used, the data used for disassembly is backed up as many acquisition sequences as there are *DECRs.

The *DECR system call only affects the channel groups that serve as inputs to the current table. If the current table has no channel inputs (only passed bits) then *DECR has no effect.

*DECR is useful when a called table (that uses group inputs) determines that the data word in acquisition memory is not what was expected, i.e., that it was not part of the previous instruction. When this occurs, *DECR gets the previous word in acquisition memory. The unexpected data word can then be disassembled correctly, starting from the next master table call. Figure 7-18 shows one possible use of the *DECR system call.

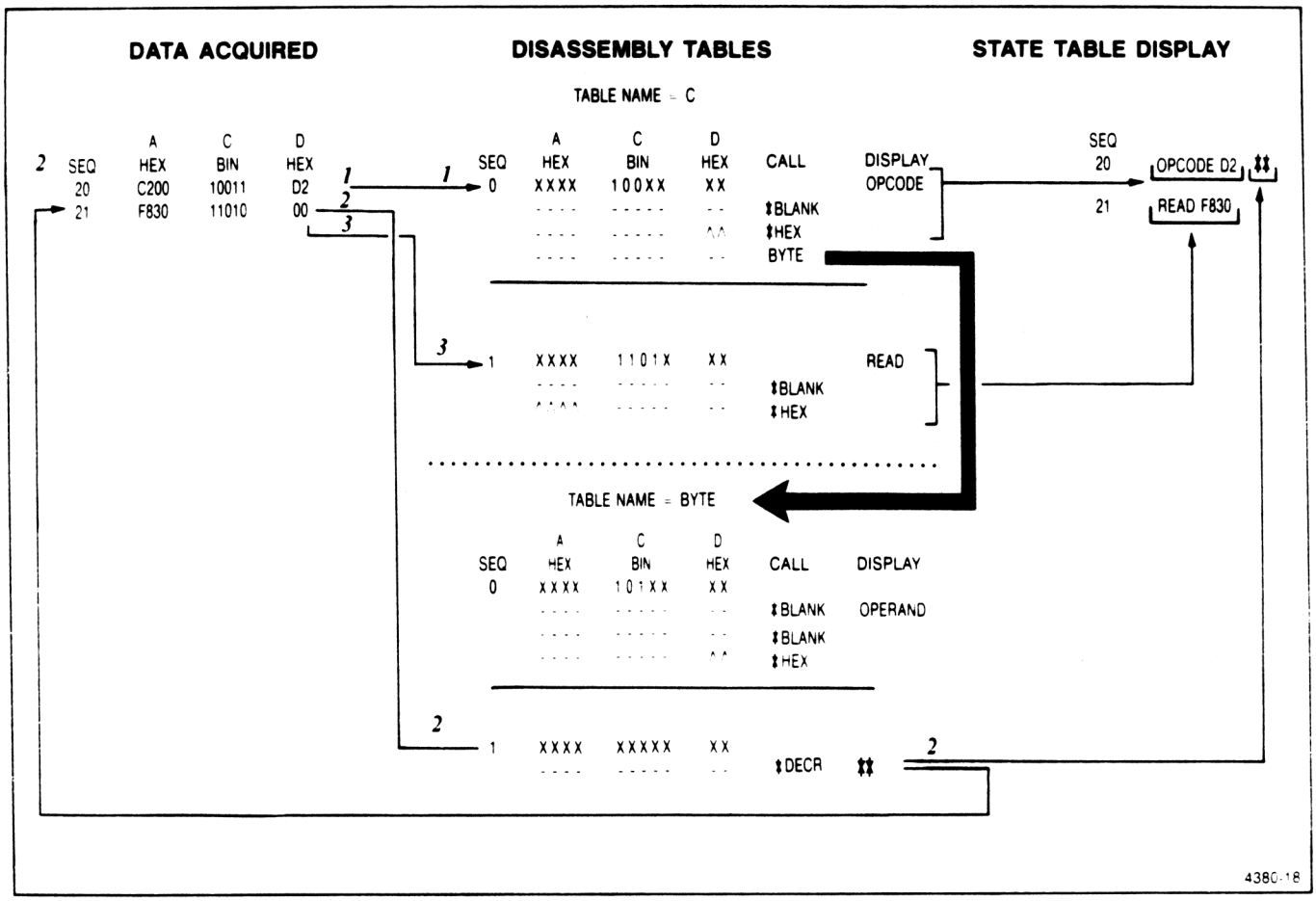


Figure 7-18. Basic operation of *DECR. In step 1, acquired data sequence 20 is sent to the master table (C) and is decoded as an opcode. Sequence 0 of table C then calls table BYTE to display the operand.

In step 2, table BYTE accesses acquired data sequence 21. However, table BYTE detects that the acquired data on sequence 21 is a read cycle not an operand. Since the data was not an operand, table BYTE prints **, performs a *DECR, and returns disassembly to the master table (C).

In step 3, the data acquired on sequence 21 is sent to table C where it is decoded as a read cycle and displayed.

System Calls for Pipelined Processors

These system calls are intended to faciltate disassembly with pipelined processors.

- The system calls *NOP, *RECALL, *LOOP, and *SKIP are useful when determining whether an instruction that enters the pipeline is actually executed.
- The *MARK and *MIRACLE system calls are useful for changing the order in which data is displayed.

Details of these system calls are provided in the following paragraphs.

NOTE

*System calls *MARK and *MIRACLE only work if the master table for disassembly is table C. They have no effect when any other master table is used.*

**MARK, *MIRACLE, and *SKIP have no effect if you scroll backwards through the State Table display. However, by pressing the STATE TABLE key after scrolling backwards, you can display the mnemonics with all system calls operating.*

***SKIP.** The *SKIP system call prevents the disassembly of the current machine instruction from being displayed. Disassembly of the machine instruction continues until completed, but the results are not displayed. Disassembly display is enabled again when the disassembly of the current machine instruction is completed and the master table is re-entered.

*SKIP is useful if, for example, a table discovers that the machine instruction being disassembled was never executed by the microprocessor due to a queue flush.

***RECALL.** This system call is designed to search ahead through acquired data. When a *RECALL occurs, the table containing the *RECALL calls itself (therefore receiving the next acquired data), and a counter is incremented to show how many times the table has *RECALLed itself. The table may *RECALL itself through the entire acquisition memory. The table may have group inputs or pass bits to itself by passing bits to the *RECALL.

NOTE

**RECALLs that search far through acquisition memory may take a long time to display results. If a disassembly routine with a *RECALL takes a long time to display mnemonics on the State Table, you can press the STOP key. Pressing STOP stops disassembly of the current word and restarts disassembly on the next acquired word.*

When the table containing the *RECALL is finished (reaches the end of a table sequence), the table performs as many *DECRs as there were times that the table *RECALLed itself, before returning to the calling table.

The end result is that even though the *RECALL may have scanned through the entire acquisition memory, the stack does not overflow. Also, to any tables that receive group data after the *RECALL is finished, it appears that the table with the *RECALL only looked at the first word passed to it—not at any of the following acquired data.

For example, a table with a *RECALL might look ahead through memory to see if an instruction going into a pipelined processor was actually executed. If the instruction was executed the disassembly would be displayed. If not, a *SKIP would be called. When the table with the *RECALL was done, the disassembly would continue as though the table with the *RECALL only looked at one word, not several.

Place *RECALLs at the end of table sequences. After a *RECALL is executed the mnemonic table does not return to complete any disassembly placed after the *RECALL.

NOTE

*If *RECALL is used in a table that receives passed bits, the *RECALL must have as many bits passed to it as the table expects. Otherwise, the error message INCORRECT NUMBER OF BITS PASSED will appear during disassembly.*

Figure 7-19 shows how *RECALL might be used to locate data far down in the acquisition memory and return to the current disassembly status.

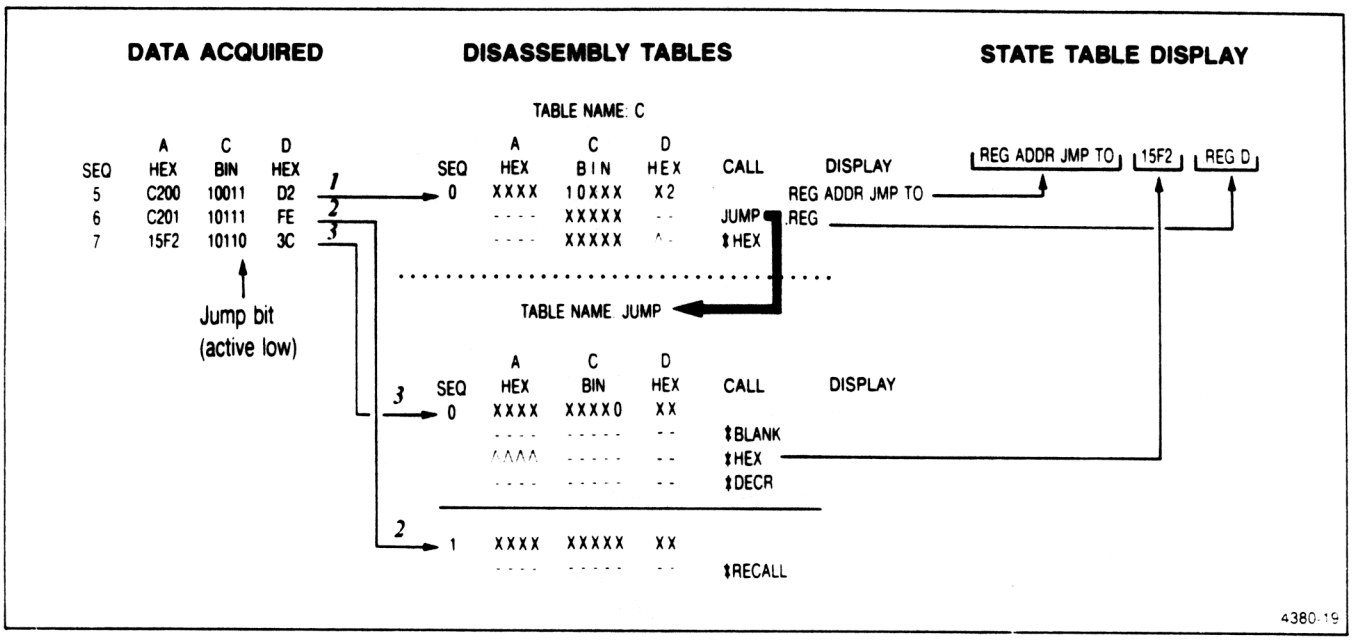


Figure 7-19. Using *RECALL. In this example the user is trying to decipher a register-addressed jump. In step 1, acquired data sequence 5 is sent to table C. Table C decodes the data as a register-addressed jump. Since the register is internal to the processor, table C must use table JUMP to search through acquired data and discover where the jump is going.

In step 2, table JUMP receives acquired data sequence 6, but finds that this sequence does not contain the jumped-to address. Table JUMP performs a *RECALL so it can look at acquired data sequence 7.

In step 3, table JUMP receives acquired data sequence 7, which contains the jumped-to address. Table JUMP displays the jumped-to address on the screen and returns disassembly to table C. The combination of *RECALL and *DECR in table JUMP sets the acquired data so table C receives acquired data sequence 6 next. So the next table to access acquired data will be unaware that table JUMP ever looked at the acquired data.

***LOOP.** This system call causes the table to loop on itself without overflowing the stack. *LOOP works similarly to *RECALL, but does not perform any of the automatic *DECRs that *RECALL does when returning to the calling table.

NOTE

*If *LOOP is used in a table that receives passed bits, the *LOOP must have as many bits passed to it as the table expects.*

***NOP.** This system call is used to rearrange the order in which bits are displayed or used. *NOP is only useful when followed by *RECALL, *LOOP, *MIRACLE, *HEXS, *HEX, *OCT, or *BIN system calls.

*NOP allows you to gather specific bits from more than one line of a call table sequence, which you can then pass to an appropriate system call. When you use *NOP to pass bits to *RECALL, *LOOP or *MIRACLE, you must gather the exact number of bits that the table expects. Figure 7-20 shows how *NOP can be used to rearrange the data display. Figure 7-21 shows how *NOP and *RECALL can operate together.

NOTE

In Figure 7-21 the user knew the table would receive 001 as bits passed from the previous table. Passing unknown bits would cause unpredictable results.

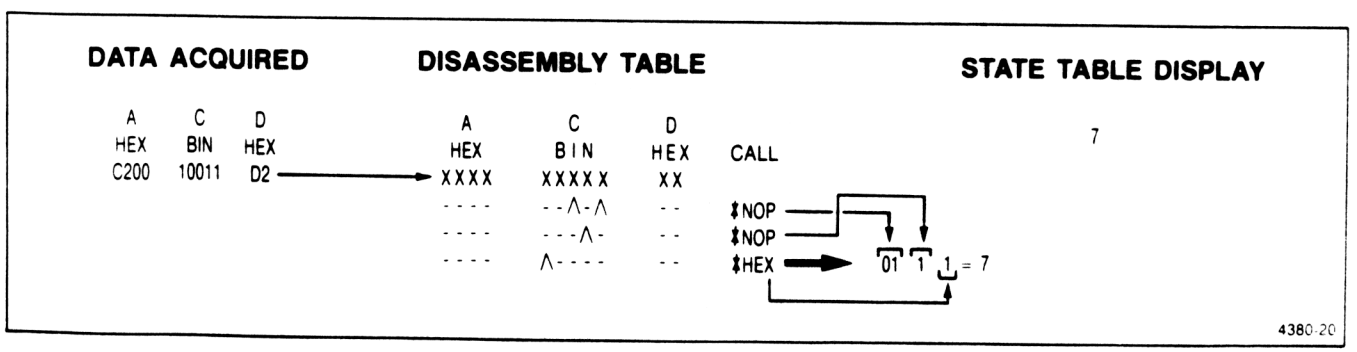


Figure 7-20. Using *NOP and *HEX to rearrange data. In this example, the user rearranged control bits to make a hexadecimal code.

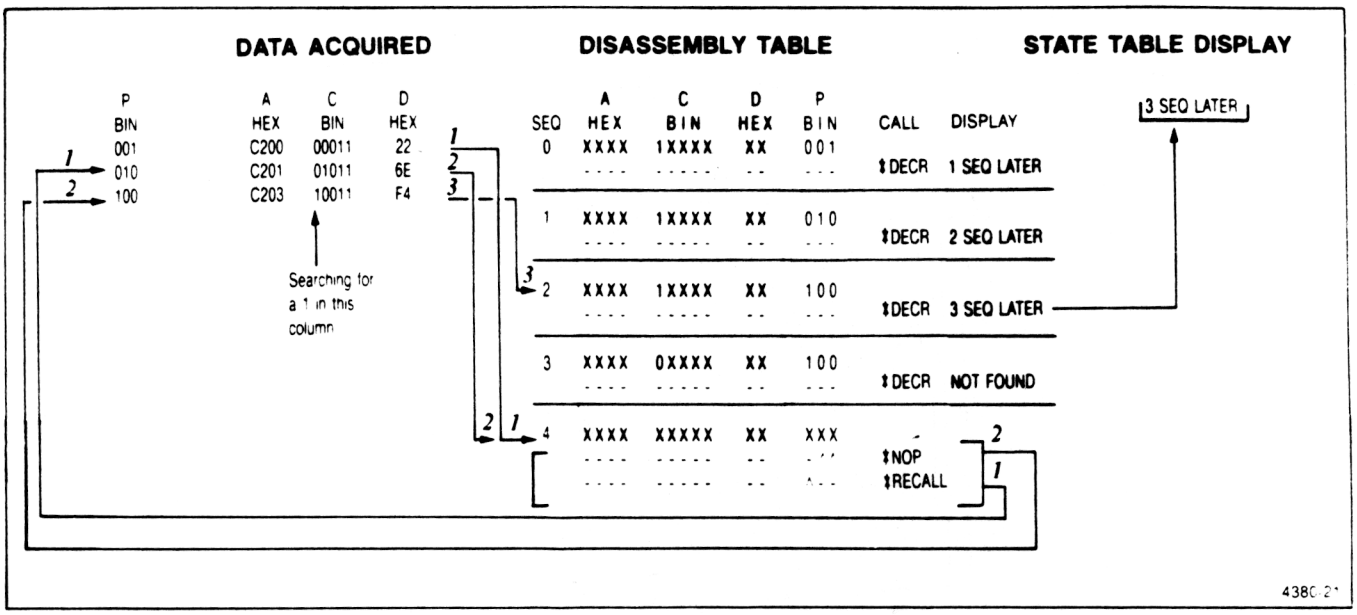


Figure 7-21. Using *RECALL with *NOP. This table would be used to search for an occurrence later in acquired data. The user designed all tables which call this table to always pass 001. SEQ 4 of the disassembly table acts as a shift register on the bits passed to indicate how many *RECALLS have been executed. In step 1, the data received by the table does not match the searched- for pattern, so sequence 4 of the table uses *NOP and *RECALL to do a shift right on the passed bits and look at the next acquired data.

In step 2, the next acquired data is received by the disassembly table. The searched for-pattern is still not found, so table sequence 4 does another shift right on the passed bits and *RECALLs the table to look at the next acquired bits.

In step 3, the last acquired data is received by the disassembly table. Disassembly sequence 2 recognizes the desired pattern, and displays a message. If the pattern sequence had not been present in step 3, disassembly sequence 3 would have displayed NOT FOUND and stopped the search. This way, a maximum of three acquisition sequences are checked, rather than the entire acquisition memory.

NOTE

*System calls *MARK and *MIRACLE only work if the master table for disassembly is table C. They have no effect when any other master table is used.*

***MIRACLE.** This system call allows disassembly of microprocessors that interweave instructions with data fetches for the previous instruction (such as the 8086).

For example, suppose a table were disassembling an instruction and it discovered a data fetch embedded in the middle of the instruction. The *MIRACLE system call stops disassembly of the machine instruction without displaying any of the results. *MIRACLE disassembles and displays the embedded instruction (the data fetch) by sending it to the master table. After the embedded instruction is displayed, then the disassembly of the original instruction is completed and displayed following the embedded instruction.

The effect of the *MIRACLE is to display the disassembly of an embedded instruction cycle in front of the instruction currently being disassembled. Figure 7-22 gives a simplified example of how *MIRACLE might be used.

The *MIRACLE system call can be used to disassemble several embedded instructions within a machine instruction, but each embedded instruction may only occupy one acquisition sequence. For example, data reads and writes will be disassembled properly by *MIRACLE, but instructions with following operands will not.

NOTE

*If embedded instructions decoded with the *MIRACLE call occupy several acquisition sequences, erroneous data may be displayed.*

*If *MIRACLE is used in a table that receives passed bits, you must pass the *MIRACLE the same bits that the table received. For example, if your table recieves group inputs A, C, and D and 3 passed bits, the *MIRACLE should look like:*

A HEX	C BIN	D HEX	P BIN	
[- - - -]	[- - -]	[- -]	[∧∧∧]	*MIRACLE

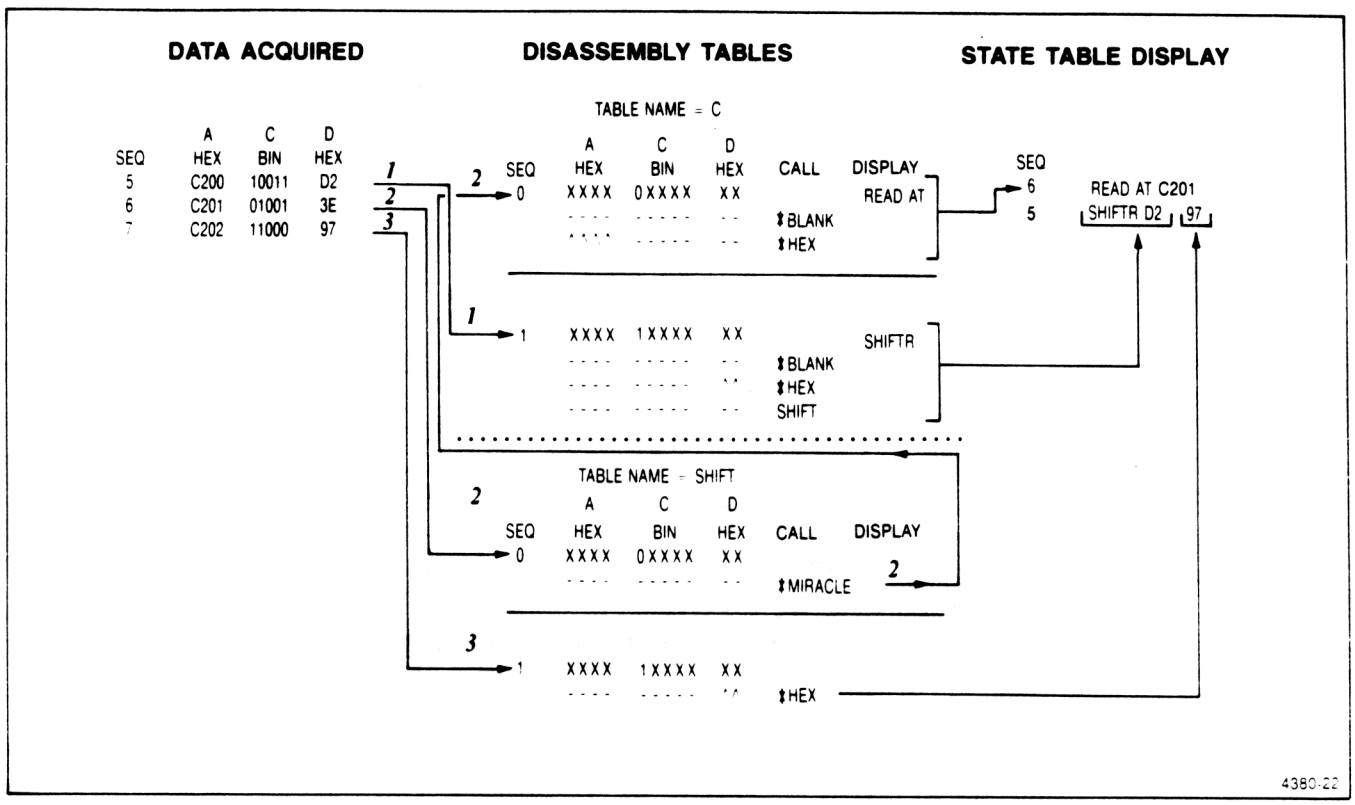


Figure 7-22. Using *MIRACLE. This illustration shows a simplified microprocessor with two instructions: READ and SHIFTR. The SHIFTR instruction requires two machine cycles to execute. These two machine cycles may be separated by another bus cycle.

In step 1, the master table (C) detects the first cycle of a SHIFTR and calls table SHIFT to read the next half of the instruction.

In step 2, table SHIFT recognizes that the next cycle is not a SHIFTR, so it performs a *MIRACLE. The *MIRACLE sends acquired data sequence 6 back to table C where the READ instruction is disassembled and displayed, completing the *MIRACLE.

In step 3, acquired data sequence 7 is sent to table SHIFT because of the *MIRACLE in the previous step. Table SHIFT now completes the disassembly of the SHIFTR instruction. The entire SHIFTR instruction is displayed on the State Table line below the READ instruction.

***MARK.** This system call lets you mark an acquisition sequence for display after disassembly of the current machine instruction is completed. This allows disassembled data to be displayed in the proper order. You may have up to three *MARKed acquisition sequences in your disassembly at any one time.

*MARK is useful for showing all of an instruction before any of the associated reads and writes are displayed. For example, when a 68000 microprocessor in the absolute long addressing mode performs a move operation, the source memory read occurs before the entire instruction is fetched. You can use *MARK to display the memory read after the instruction operands are completely disassembled.

Figure 7-23 shows how the *MARK system call might be used.

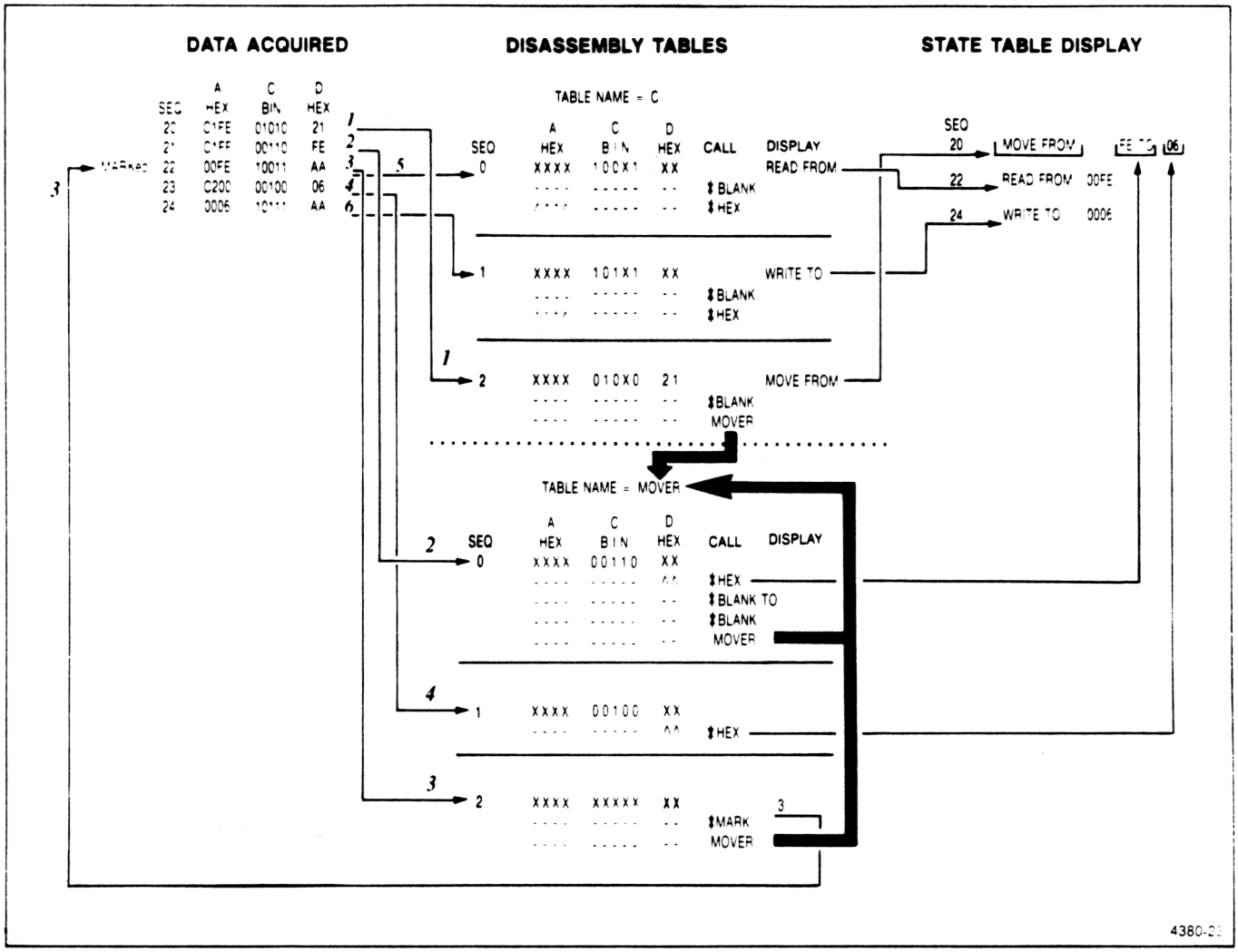


Figure 7-23. Using *MARK. This example shows a processor with three instructions: READ, WRITE, and MOVE. Move has two operands, a FROM operand and a TO operand, and the processor does a read between the two. The processor starts executing a MOVE instruction before the entire instruction is fetched.

In step 1, table C detects a MOVE opcode, displays MOVE FROM, and calls table MOVER to get an operand.

In step 2, table MOVER reads data sequence 21, decodes the data as a FROM operand, and calls MOVER again to get the last operand.

In step 3, table MOVER reads data sequence 22. Sequence 22 is not a MOVE operand, however, so sequence 2 of table MOVER *MARKs the acquisition sequence and calls table MOVER again to look at the next acquired word.

In step 4, table MOVER receives data sequence 23. Sequence 1 of table MOVER identifies the last operand of the MOVE instruction and displays it.

In step 5, disassembly returns to the *MARKed instruction. The master table (C) receives data sequence 22 and decodes it as a READ instruction.

In step 6, disassembly of the MOVE and the *MARKed instruction is now complete. Disassembly goes to acquisition sequence 24, which table C decodes as a WRITE instruction.

DISASSEMBLY GUIDELINES

STUDY YOUR PROCESSOR

The best way to determine the disassembly tables you need is to carefully study your processor. The tables described below are suggested as a starting place. You will probably need tables in addition to those described below to cover special requirements of your processor.

Acquisition Channels. To insure compatibility with all system calls, you should define your table structure so that the following signals are acquired through the correct Channel Specification groups:

- acquire all address bus signals through group A
- acquire all control signals through group C
- acquire all data bus signals through group D

Master Table. This CALL-type table is where disassembly starts. Table C is recommended as the master table. If the acquisition channel recommendations are followed, then groups A, C, and D should be the group inputs.

The master table must have group inputs from all groups that acquire disassembly data. Under most circumstances these are the groups that acquire control signals, data, and address. The master table determines whether the acquired data is an opcode, and then passes the acquired bits to the appropriate table. If the data is an opcode, table OPCODE is called; otherwise table ABSOLUTE is called.

Before calling tables OPCODE or ABSOLUTE, the master table should display the acquired address, using the *HEX system call, so the address bits do not have to be passed to later tables.

ABSOLUTE Table. This table is needed to decode all processor states other than opcode fetches. The master table passes data bus and control signal bits to this table. The ABSOLUTE table examines the control signals to determine the operation performed by the processor, then displays the operation type. Usually, the ABSOLUTE table displays occurrences like memory reads, memory writes, I/O reads, I/O writes, and interrupt requests.

OPCODE Tables. Because most processors have a large instruction set, opcode decoding usually requires several tables. When the master table determines that an opcode was acquired, it passes bits to table OPCODE. The bits passed to OPCODE are those needed to decode the opcode; usually all control signals and the bits from the data bus. With small instruction sets, table OPCODE can decode the entire instruction set. Usually the instruction set is too large, so table OPCODE divides the opcodes into major categories. These categories are determined by significant bit patterns in the opcode. Possible categories are:

- opcodes that have no following operand
- opcodes that use registers
- opcodes that have one operand
- opcodes that have two operands
- condition-testing opcodes like branch or jump

There are other possible categories, depending on your processor and the bit patterns that its opcodes follow. Each category has a corresponding table, named OPCODE2, OPCODE3, etc. Table OPCODE tests bits to find the proper category, then passes all the necessary bits to the corresponding table. The second opcode table then completes disassembly and displays the results.

Opcode tables can call support tables which display operands or reduce table size. All opcode tables must be of the CALL type to use support tables. Commonly used support tables are discussed next.

Address Mode Table. Processors with several address modes usually require this table. The table is called by an opcode table to provide punctuation for an operand (such as #, spaces, commas, or parentheses).

The address mode table receives bits from the opcode table, displays punctuation in front of the operand, then passes bits to an operand display or register table. When disassembly returns from the operand display or register table, the address mode table adds any closing punctuation and returns to the opcode table.

Operand Display Tables. Operands follow an opcode in the data stream, therefore operand display tables must access words following an opcode. Operand display tables might be needed to display one word operands (with a table called BYTE), or two word operands (with a table called WORD).

Operand display tables must access the control signal and data groups, group inputs C and D, but do not usually receive passed bits. Opcode tables determine when and where an operand must be displayed, and then call an operand table. Examples of typical operand display tables are given in Figures 7-12, 7-13, and 7-14.

Register Tables. Bit patterns indicate particular registers in a microprocessor. Since the patterns are consistent from one opcode to the next, one register table can decode registers for many opcode tables. This reduces the size of the opcode tables and speeds disassembly. If there are several register sets or several bit patterns which access the same register, then more than one register table may be required. For example, some processors may need a REG8 table to decode 8-bit registers and a REG16 table to decode 16-bit registers.

Opcode tables use register tables by passing them the register indication bits. The register table displays the register name that matches the passed bit pattern and returns disassembly to the opcode table. Since the register table does not call other tables it can be a DEFAULT-type table, unless system calls are used.

Error-Handling Tables. Sometimes an illegal opcode or illegal control line state is acquired. In these cases use the *ERROR system call and create table ERROR* to indicate errors on the state table screen. The ERROR* table can then be called using *ERROR whenever an error is detected.

Call table structures in the Define Mnemonics menu should be designed carefully. Attention to detail will prevent problems such as improper disassembly, endless loops, or truncated mnemonics. The remaining text in this section describes where possible problem areas exist and what to do if problems occur.

USING THE DISPLAY SETUP SUB-MENU WITH CALL TABLES

The Display Setup sub-menu controls which data and mnemonics appear on the State Table display and how wide each display is. The following rules will be helpful when controlling the display of call table mnemonics with this menu:

1. Characters displayed by call tables are cumulative. Be sure the MNEMONICS WIDTH field for your master table has a large enough value to show all your mnemonics. Mnemonics that are too long are truncated from the right.

2. If you need more State Table display area, set the DISPLAY DATA field to NO. In many applications, the numerical data acquired is not useful. In these cases only the DISPLAY MNEMONICS field should be set to YES.

3. If you use *MARK or *MIRACLE, and you have more than one possible master table in your call table structure, do not display mnemonics from table C along with mnemonics from any other table. Doing so may cause incorrect disassembly.

DESIGNING CALL TABLE STRUCTURES

The following are recommended design practices for using the Table Definition and the Table Entry sub-menus to create a call table structure. These recommendations are intended to provide consistent disassembly results.

There may be situations when the recommended practices will not produce the disassembly you want. In these cases, experiment freely. It is not possible to disrupt the mnemonic structure, any DAS setups, or the acquired data with a peculiar table structure.

NOTE

If disassembly is taking a long time or is in an endless loop, pressing the STOP key will stop disassembly of the current data and move disassembly on to the next acquired data.

Choose the Correct Groups and Tables

For almost all disassembly situations you will want to acquire your address through group A in the Channel Specification menu. The *RLADD system calls assume that group A contains the address of the data acquired.

Use table C as your master table if you are going to use either *MARK or *MIRACLE. These system calls will not work with any other master table.

Always acquire some data for disassembly through the group that corresponds to your master table. For example, if you acquire data through groups A, C, and D, use only table A, table C, or table D as a master table.

Design Complex Disassembly Structures on Paper First

Once a table is defined and entered into the DAS, some of its characteristics are difficult to change. After a table with bits passed or group inputs has a sequence entered, the number of bits passed and the group inputs to that table cannot be changed. To change the group inputs or the number of bits passed to a table, all sequences must be deleted from the table. This is indicated in the Table Definition sub-menu when the SEQ COUNT field for that table is zero.

The Define Mnemonics menu will nest call tables 16 deep. If the disassembly routine calls too many tables, the error message TOO MANY CALLS will appear on the State Table display. Sometimes using *RECALL or *LOOP can reduce the number of calls nested.

Conserve Tables and Table Entries

There are 48 user-definable tables. For almost all applications these will be more than enough tables. Still, judicious use of tables will leave room for changes or expansion in your disassembly routine.

Defined tables require space in the DAS memory. Overly large table structures may exceed the amount of memory available for the Define Mnemonics menu. In these cases the DAS will display the error message TABLE FULL.

If the TABLE FULL message appears, and a reference memory has been stored, you can create more memory space for your tables by reducing or removing the reference memory. Change the size of the reference memory by changing the COMPARE: START SEQ and STOP SEQ fields in the Reference Memory sub-menu of the State Table menu and pressing the STORE key. The Define Mnemonics menu and the reference memory share the same memory block. By reducing the size of the reference memory, more memory is available for disassembly tables.

Use System Calls Carefully

The most common system calls, like *HEX and *TAB, present no difficulty for the user. There are six system calls that deserve special attention:

- *DECR, if used improperly, may cause the disassembly to go into an infinite loop. For example, *DECR followed by *RECALL will call the same table with the same data until you interrupt the loop by pressing the STOP key.

- *RECALL can cause a search through the entire acquisition memory. If the first occurrence of the searched-for data is not present in the acquisition memory, the *RECALL may find a second occurrence and display it, which would cause incorrect disassembly. This problem could occur when disassembling data that was acquired with qualifiers operating.

  When *RECALL is used in a table that receives passed bits, you must pass the same number of bits to the *RECALL as the table expects. For example, if a table with a *RECALL is defined in the Table Definition sub-menu as receiving five bits, then you must pass five bits to the *RECALL in that table. If you do not pass the correct number of bits to *RECALL, the error message INCORRECT NUMBER OF BITS PASSED will appear during disassembly.

- *LOOP requires the same precautions that *RECALL has.

- When *MIRACLE is used in a table that receives passed bits, you must pass the same number of bits to the *MIRACLE as the table expects. If you do not pass the correct number of bits to *MIRACLE, the error message INCORRECT NUMBER OF BITS PASSED will appear during disassembly.

  *MIRACLE only disassembles data that was acquired on one sequence. If the *MIRACLE call to the master table decodes data from several acquisition sequences, erronious data may be displayed.

  *MIRACLE will also only operate in a table structure with table C as the master table. If table C is not the master table, the *MIRACLE is ignored.

- *MARK will only operate in a table structure with table C as the master table. If table C is not the master table, the *MARK system call is ignored.

- *MARK only disassembles data that was acquired on one sequence. If *MARKed data is decoded over several acquisition sequences, erronious data may be displayed.

- *HEXS only works on 8 or 16 bits. If the wrong number of bits is passed to *HEXS, negative word values can cause the wrong number to be displayed.

Keep Group Data Aligned

Data acquired through different groups in the State Table menu can become misaligned when accessing the next data. This misaligned data may result in faulty disassembly which is very difficult to detect. Misaligning data intentionally can be useful when reducing table sizes or decreasing the number of nested calls. Read the following paragraphs carefully before intentionally misaligning data between groups.

Normally, group data will remain aligned if you use the following rule:

> When accessing a series of data words, do not access a group that the previous tables in the call sequence have not accessed. For example, if table X accesses groups C and D and then calls table Y, table Y should access only groups C and/or D. If table Y accessed group A, the group data would be misaligned.

The Define Mnemonics menu uses a set of pointers into memory to indicate which data it has used. There is one pointer for every group accessed by the master table. For example, if groups A, C, and D are accessed by the master table, three pointers are set up; one for each accessed group.

The pointer for a group is advanced whenever the Define Mnemonics menu uses a word from that group. Calling a table that accesses group C advances the pointer for group C, but leaves the other pointers as they were. Figure 7-24 shows how accessing groups in the wrong sequence can misalign group data.

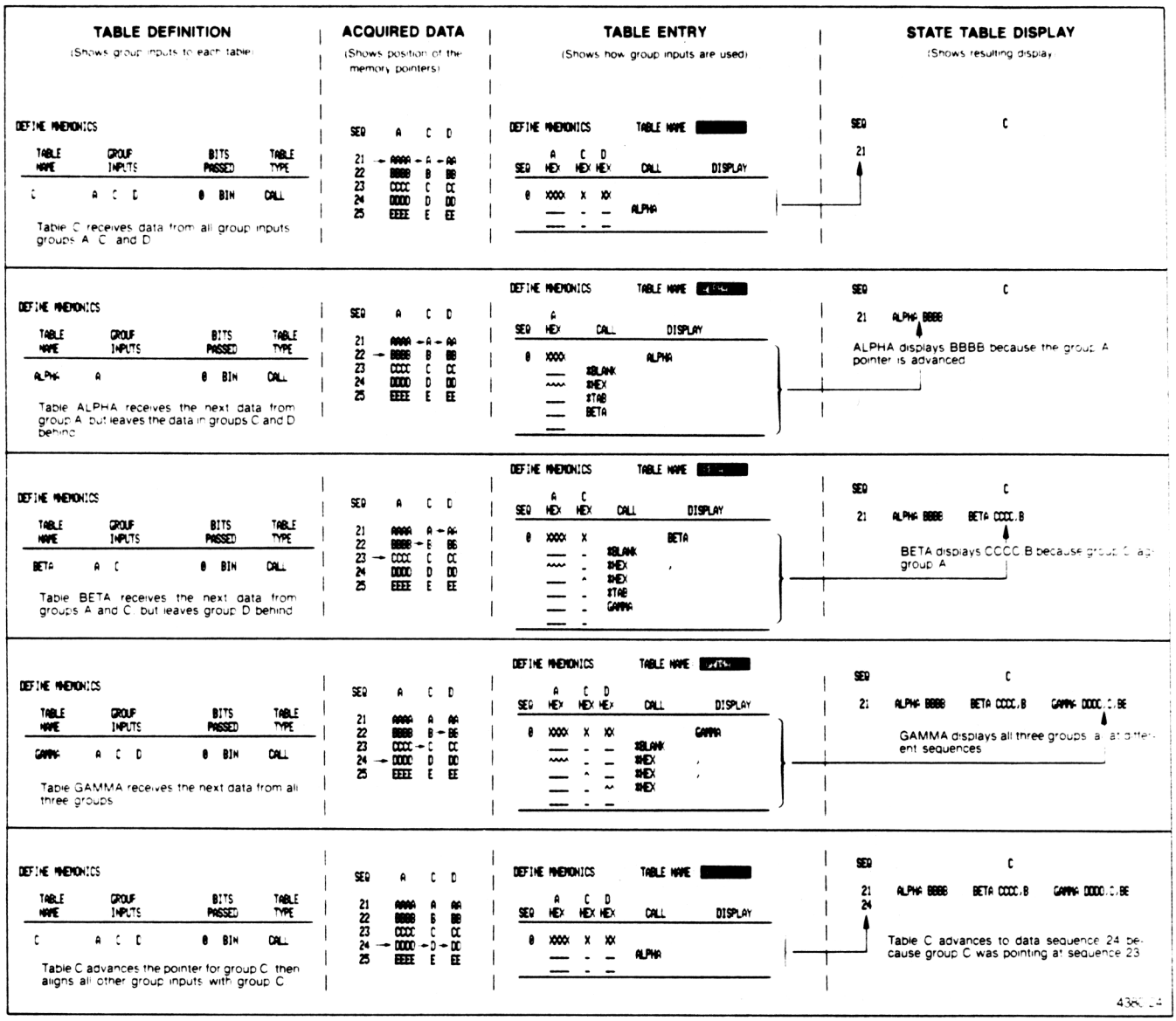


Figure 7-24. How improper group access can misalign display of group data with call table disassembly. Display of data is offset because the called tables do not access all of the groups that are used by tables further in the call sequence.

Whenever the master table is called to start disassembly of a new instruction, all pointers are set to the same position as the pointer for the master table's group. For this reason, if you do not want to see the same data disassembled more than once, you should always access the master table's group whenever you are accessing any group data.

Any table with group access needs to access:

- the groups the called table operates on
- any groups that tables called later will operate on

This will prevent the group pointers from being misaligned. In most cases a table with group access should also access:

- the master group

Otherwise, you may see the same data disassembled more than once.

Keep in mind that the pointers are only realigned when the master table is called. The problems indicated in Figure 7-24 may still occur between times when the master table is called.