AMT

DAP Series

APAL Language

(man005.03)

Technical publication man005.03

First edition June 1987
Second edition October 1987
Third edition 3 October 1988

AMT will be pleased to receive readers' views on the contents, organisation, etc of this publication. Please make contact at either of the addresses below:

Publications Manager
Active Memory Technology Ltd
65 Suttons Park Avenue
Reading
Berks, RG6 1AZ, UK

Tel: 0734 661111

Publications Manager
Active Memory Technology Inc
16802 Aston St Suite 103
Irvine
California, 92714, USA

Tel: (714) 261 8901

Preface

This manual describes the Array Processor Assembly Language (APAL), which represents the lowest level at which a DAP-series machine can be programmed; the manual covers the use of APAL with all models in the DAP-series range.

Although this manual is a definitive reference for the syntax of APAL, chapters 1 to 12 are intended to be read serially as an introduction to the language (although you should preferably have some experience of assembly language programming in general); appendix F describes the APAL instruction set in alphabetical order. After you have studied chapter 1, you may find it helpful to look at the examples in appendix C, to get an impression of the style and level of programming of APAL before you continue with the detailed description in chapter 2 and onwards.

Descriptions of features of the language are given in two parts: the first part is a formal syntactic definition of the language feature, and the second part is a description of its semantics.

Chapter 1 describes the architecture of a DAP-series machine. Chapter 2 describes the format of APAL source text. Chapter 3 describes the structure of an APAL source module. Chapter 4 describes the declaration and initialisation of data. Chapter 5 describes the declaration of code sections. Chapter 6 describes the APAL instruction set. Chapter 7 describes the various modes of addressing possible in APAL. Chapter 8 describes the run-time tracing facility. Chapter 9 describes code section conventions and mixed language programming with FORTRAN-PLUS, the superset of FORTRAN 77 implemented on DAP-series machines. Chapter 10 describes miscellaneous assembly-time facilities such as listing control. Chapter 11 describes assembly-time facilities such as conditional assembly and textual substitution. Chapter 12 describes the APAL macro facility.

The preparation of a DAP program involving APAL routines is described in the AMT manual *DAP Series: Program Development*; consult the version of the manual appropriate to your host system.

Other relevant AMT manuals are:

DAP Series: Introduction to FORTRAN-PLUS	(man001)
DAP Series: FORTRAN-PLUS Language	(man002)

Syntax and semantics

Both the syntax and the semantics of the various APAL code statements are covered in the manual at the appropriate places. To get a full understanding of how to use a particular statement you need to read *both* the syntax and semantics sub-sections covering the statement.

The formal syntax of APAL is defined using Backus-Naur form (BNF), with the following extensions:

- <entity >? means that <entity > is optional; <entity > may occur once, or not at all
- <entity >* means that <entity > may occur several times, or not at all

As usual with BNF notation, items printed in capitals in the syntax (*terminals*) are to be included in your code exactly as they are stated in the syntax; items printed in lower case and enclosed within '< >' – such as '<entity>' above (*non-terminals*) – are to be replaced in your code by an appropriate terminal.

In the semantics of a statement, terminals are also printed in capitals, non terminals in italics.

For example, the semantics of the data header statement is given in section 4.1 as:

> DATA *section-name* *name-property* *common-property* *write-property*

An actual data header you might have in your APAL code might be:

> DATA FREIDA HOST COMMON WRITE

where:

> FREIDA is the name of your data section
>
> You are specifying a name property of HOST; that is that you want the data to be available for reference in a call in a host program interface subroutine
>
> You are specifying the common-property COMMON; that is that the data in the section is to occupy a common area in store with other data sections in different code modules, but which have the same data section name and have the COMMON property
>
> You are specifying that you want write as well as read access to the data

See chapter 4 for fuller details of data headers!

Contents

Chapter 1

The architecture of DAP-series machines

This chapter describes the architecture of the whole range of DAP-series machines, and in particular those components of the architecture that are relevant to a programmer using the *Array Processor Assembly Language* (APAL).

1.1 The DAP series

The AMT DAP series of machines is the latest generation of a parallel processing architecture based on a 'distributed array of processors' (DAP). A DAP-series computer is connected to its host machine as an attached processor. A simulation system is also available, which enables you to develop and run programs in the absence of DAP-series hardware.

(In order to help you make easy reference to the accompanying figures, the text starts again on the next page.)

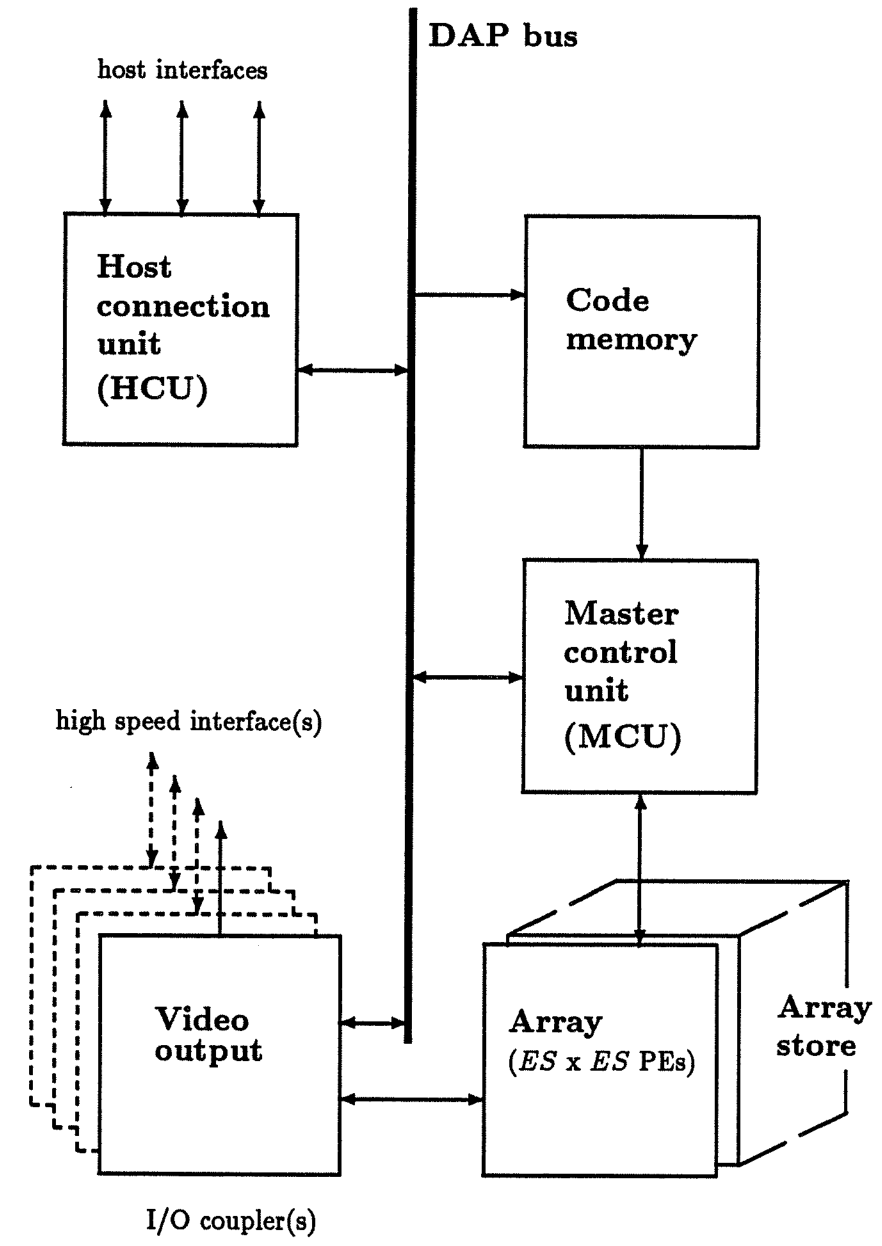

Figure 1.1 Block diagram of a DAP-series machine

Figure 1.1 is a block diagram of a DAP-series machine. The backbone of all DAP-series machines is the *DAP bus*, an internal synchronous 32-bit multiplexed bus, into which the various component parts of the machine are connected.

The main computing resource in the DAP is the *array* or matrix of *processor elements* (PEs), which is closely connected to an *array store* that holds the data to be processed. A DAP-series machine achieves its high performance by having each processor element operate in parallel with all other processor elements; each element operates on its local data, subject to local (or *activity*) control when required.

The processors are arranged in a square matrix; the number of processors on one side of the square gives the *edge size* of the machine; AMT uses *ES* to signify edge size. The different ranges of machines in the DAP series are characterised by the value of *ES*. For the DAP 500 range, the array is a square matrix of 32 by 32 PEs – an *ES* of 32; for the DAP 600 range, *ES* is 64, and so on. Note that the edge size is a power of 2, the exponent being the first digit in the DAP model number (the other digits in the model number being the clock speed in MHz, with the memory size in Mbytes after the hyphen) so the DAP 600 range has an array size of 2^6 by 2^6 PEs, and so on for other ranges in the DAP series.

The array (the array of PEs or PE array) acts under the control of the *master control unit* (MCU). The MCU fetches and decodes instructions held in the separate *code memory*, broadcasting address and control information to all the PEs in parallel, along with data in some cases. The MCU also has facilities for integer scalar operations. Typically, the code memory is 0.5 Mbytes or more; the array store is ES^2 x 32 K-bits or more, 16 Mbytes for the DAP 600 range for example.

The interconnection to the host computer is controlled by the *host connection unit* (HCU). Connection to the host is via any one of the 3 host interfaces, with the other 2 being available for low and medium speed data input and output. Communication between the host and your DAP program is managed for you by AMT-supplied library routines (see *DAP Series: Program Development* for further details), so the presence of the HCU is invisible to you as a user.

The design of the DAP-series caters for up to 4 high speed interfaces, for video or other high bandwidth requirements, with an available bandwidth of 50 Mbytes/sec, or higher in some situations. The set of I/O couplers fitted to any particular machine will depend on customer requirements. Note that the high speed input/output capability allows you to transfer data in and out of the array store at high speed, but it does not allow high speed transfers onto the DAP bus which only supports data transfers up to a few Mbytes/sec.

A program which runs on a DAP-series machine is called a DAP program. An associated program written in a conventional language is used on the host. This host program is entered first and is then responsible for starting the execution of the DAP program, and for data transfers to and from the DAP-series machine, using special interface subroutines. The preparation of a DAP program, and the interface subroutines, are described in the AMT manual *DAP Series: Program Development*. Different versions of this manual are available for each of the host operating systems under which AMT supports DAP operation.

Facilities also exist for a DAP program to read or write host filestore, and to communicate with peripherals connected directly to the DAP-series machine; consult AMT for further details.

You can write a DAP program in APAL, which is the subject of this manual, or in FORTRAN-PLUS (see *DAP Series: FORTRAN-PLUS Language*), or in a mixture of the two. If an APAL code section communicates with FORTRAN-PLUS or a host program, there are certain conventions to be followed; these are given in chapter 9. You can write the host program in any language or mixture of languages supported by the host operating system; however, if you want to use the special AMT-provided interface subroutines, you will find things are simplified if you write the calling sections in FORTRAN. For generality, the term FORTRAN as used in this manual is intended to include FORTRAN 77.

As with many other computer systems, more than one user program can be resident in the DAP at the same time, although only one program can be running at any given time. Each user program is allocated a contiguous range of code memory addresses and a contiguous range of array store addresses. Allocation of these resources is made by the HCU, and the running of individual user programs – on a time-slice-per-user-program basis – is scheduled by a supervisor program resident in the MCU. For further details of DAP's multi-programming facilities see *DAP Series: Program Development*.

1.2 The PE matrix

Figure 1.2 opposite shows a simple conceptual diagram of a DAP. As described briefly above, the parallel processing capability of a DAP-series machine is provided by a square matrix of ES by ES processor elements (PEs), with ES taking the value 32 for the DAP 500 range, 64 for the DAP 600 range, and so on. Each PE is capable of performing arithmetic and logical operations on operands that are single bit values.

In APAL the rows and columns of the PE matrix are each numbered from 0 to $ES-1$, as shown in the figure. (In FORTRAN-PLUS they are numbered 1 to ES). The edges of the array are referred to as *North* (row 0), *East* (column $ES-1$), *South* (row $ES-1$), and *West* (column 0).

Each PE is connected to the four neighbouring PEs in the north, south, east and west directions. Using these connections, data can be shifted from a register of each PE into the corresponding register of a neighbouring PE; a north shift means that data is shifted *to* the north.

1.3 The DAP code memory

The object code of a DAP program derived from APAL or FORTRAN-PLUS source resides in a separate *code memory*. The code area of a DAP program is delineated by hardware datum and limit registers, which are inaccessible to you, the user. You are concerned only with instruction addresses relative to the datum. If you try to use instruction addresses above the value in the limit register, or negative memory addresses (below the value in the datum register), you will cause a run-time error. The actual size of the code memory depends on the actual DAP-series machine concerned, but it will hold at least 128K instructions (512 Kbytes).

1.4 The DAP array store

Each of the ES by ES PEs has a local memory whose size depends on the actual DAP-series machine concerned, but is at least 32K (= 32,768) bits. The sum total of this PE storage is referred to as the *array store*, and is at least ES^2 x 32 K-bits, 4 Mbytes for the DAP 500 range for example.

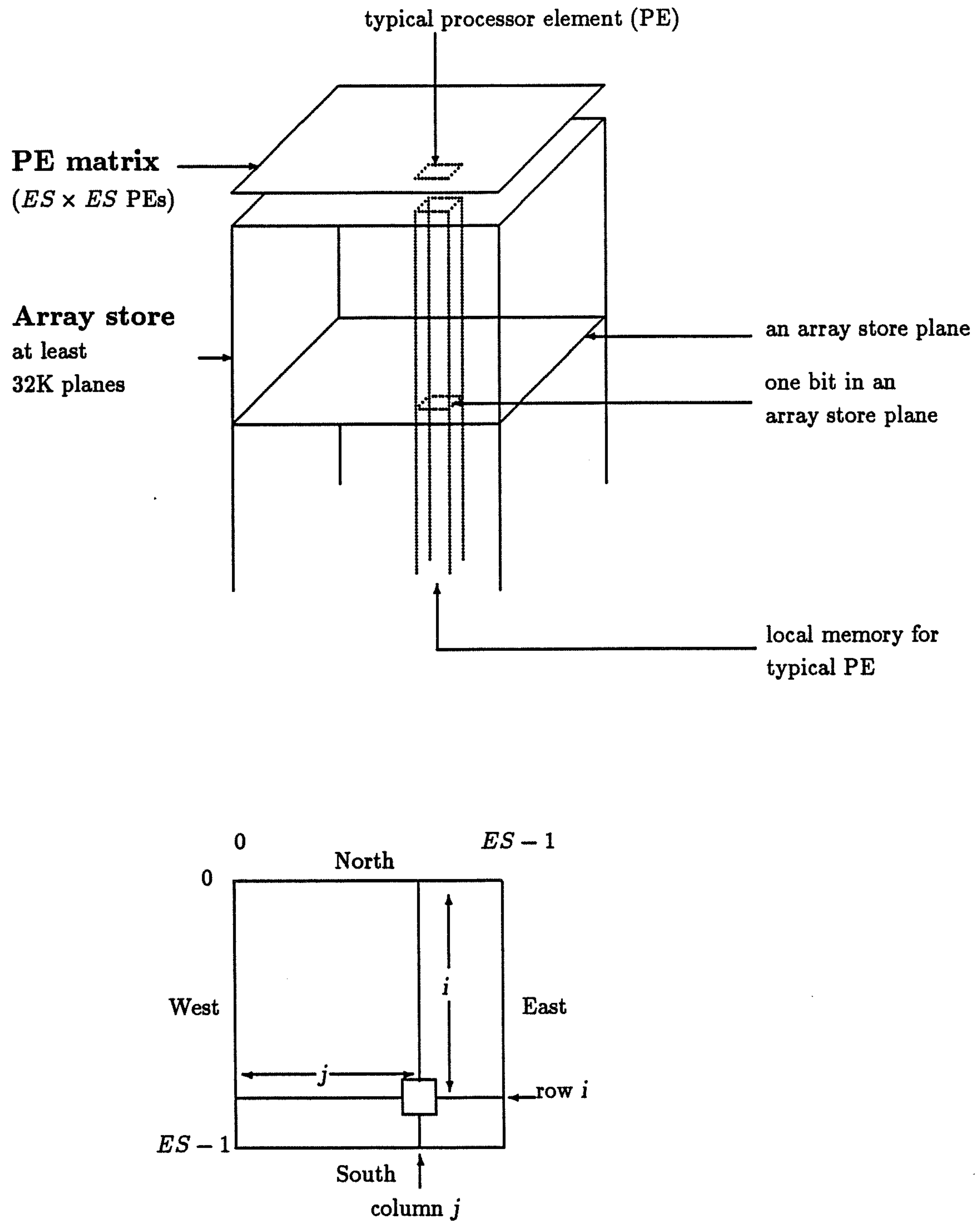

Figure 1.2 PE matrix and array store

The array store is best regarded as a three dimensional array of bits, as shown in figure 1.2, and consists of at least 32K memory planes.

A DAP word is 32 bits, so each memory plane can be looked on as ES rows of either ES bits or $ES/32$ words. Alternatively, an array plane can be considerd as ES columns of ES bits or $ES/32$ words. For example, for the DAP 600 range of machines, each memory plane can be described as 64 by 64 bits, or 64 rows of 64 bits each, or 64 rows of 2 words each, or 64 columns of 64 bits, and so on. Word 0 of each row is the most significant 32 bits of the row, so for example, on DAP 600 word 0 is bits 0 to 31 of the row, and word 1 is bits 32 to 63. The PE array maps onto the array store in such a way that the local memory for $\mathrm{PE}_{i,j}$ consists of $\mathrm{bit}_{i,j}$ of each memory plane.

Data can be transferred between the host and the array store in units of words, with one row of array store consisting of $ES/32$ words. The last (or only) word of each row is regarded by both DAP and host as immediately preceding the first word of the next row. The last word of a memory plane is regarded by both the DAP and the host as immediately preceding the first word of the next memory plane.

The array store holds the data of the DAP program. A single user program has a block of contiguous memory planes in array store; this block is delineated by hardware datum and limit registers which are inaccessible to you, the user. You are concerned only with memory addresses relative to the datum, that is, memory planes are numbered from zero, starting with the plane addressed by the datum register. If you try to access data outside the range defined by datum and limit, you will cause a run-time error.

The two blocks of code and array stores assigned to your DAP program are known collectively as the DAP program block; its structure is shown in figure 1.3.

You can use planes 0 to 119 freely as workspace. Planes 120 to 127 are reserved for the run-time system and must not be written to. Certain literal values (created implicitly by some of the APAL instructions) occupy plane 128 onwards along with any APAL or FORTRAN-PLUS data sections declared to be read-only.

The stack area is managed by software according to conventions described in chapter 9. The size of the stack area is determined during the consolidation phase; it can be controlled by parameters input by you at consolidation time (for more details see the version of *DAP Series: Program Development* relevant to your host system).

1.5 Data mapping and number representation

You control the mapping of data onto the array store. Two mappings that are particularly well suited to the structure of DAP-series machines are:

- Vector (or horizontal) mode, in which successive bits of a data item are mapped onto successive bits of a single store row
- Matrix (or vertical) mode, in which successive bits of a data item are mapped onto the same bit position in successive store planes

These mappings correspond to FORTRAN-PLUS vector and matrix storage modes respectively (see *DAP Series: FORTRAN-PLUS Language*).

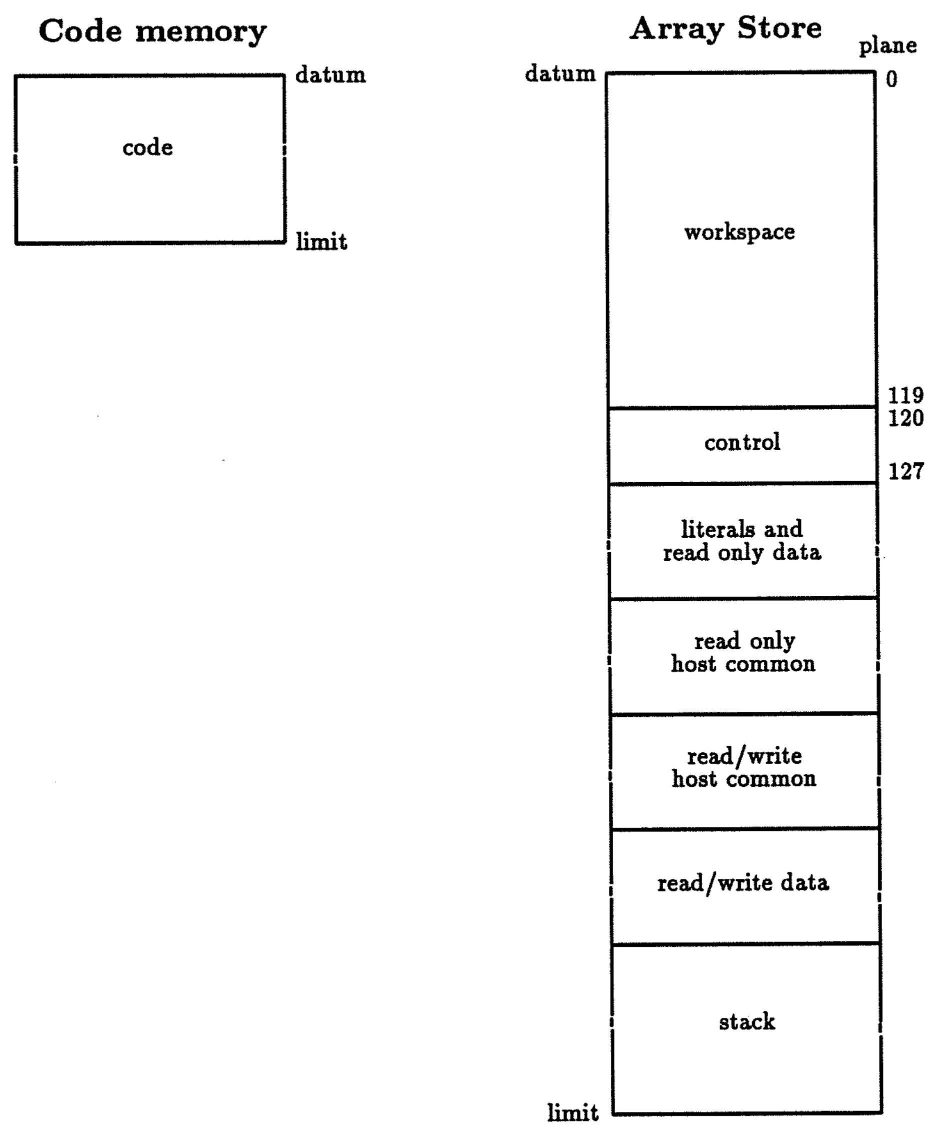

Figure 1.3 DAP program block

Because of the bit orientation of its hardware, a DAP-series machine is not committed to any particular representation of data. In general, a DAP-series machine regards data simply as arrays of bits, the interpretation of which (as fixed or floating point numbers, for example) is entirely dependent on software. However, you should note the following exceptions to this independence:

- The hardware supports arithmetic operations on 32-bit signed integers (2's complement) in the MCU registers
- The hardware supports the parallel addition of *ES* pairs of up to *ES*-bit integers held in the array

1.6 Processor element functions

Figure 1.4 is a simplified diagram of one processor element (PE) showing its main functional components.

Each PE has three one-bit registers, denoted A, C and Q, each of which can be clocked or not, depending on the instruction.

The C- and Q-registers are input to an adder. Depending on the specific instruction either or both of these inputs can be treated as False (using AND gates, not shown in the figure).

The third adder input is selected by a multiplexor and can come from the PE's memory via the S register, the outputs of the Q- or A-registers, data broadcast by the MCU in either 'row' or 'column' orientation, or the carry output of a neighbouring PE. For each of these input sources, except the neighbour carry output, there is the option of inverting the multiplexor output using an Exclusive-OR gate (not shown in the figure).

The A-register also receives its input from this multiplexor, and it may either be written in directly or AND-ed (masked) with the existing A-register contents; this gating at the input to the A- register is not shown.

PE outputs may be written to memory, and in some instructions this writing is conditional on the value in the A-register, which thus acts as an activity control. With some instructions, there is the option of writing to just one row or one word of memory rather than to the entire plane.

The D-register shown in the PE diagram does not appear explicitly in the APAL programmers model of the PE, but is used for input or output under control of the fast input/output hardware.

Instructions that perform activity-controlled write-to-memory achieve their effect by performing a memory read-modify-write sequence. Clearly a register is needed somewhere in this path; the details are implementation dependent, but an example of such a register is shown as S in the diagram; again this register does not appear in the APAL programmer's model.

1.7 The MCU

The Master Control Unit (MCU) performs the following functions:

- Instruction fetching, decoding, and address generating

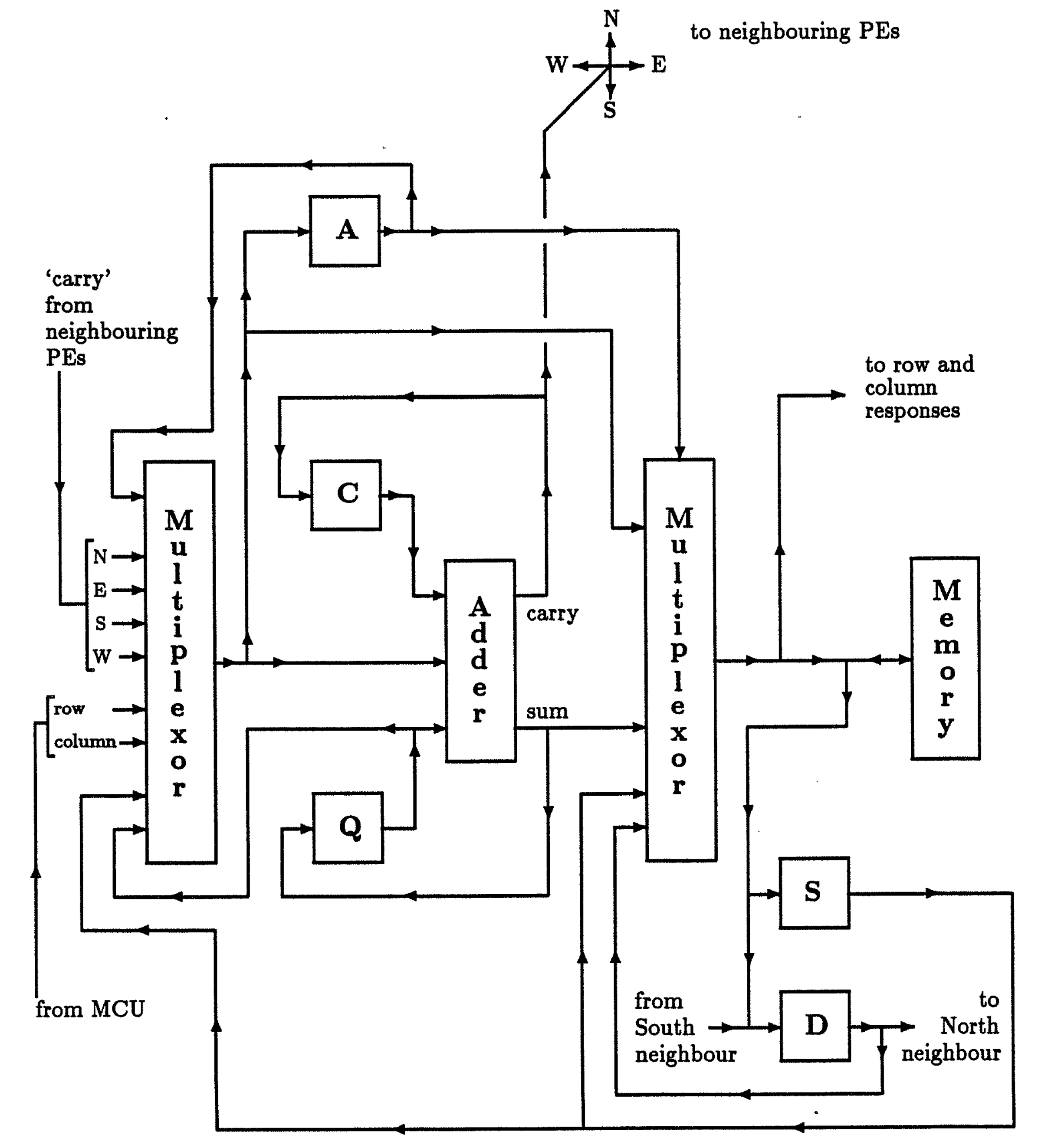

Figure 1.4 Simplified diagram of a processor element

- Executing certain instructions, and broadcasting other instructions to the PE matrix for simultaneous execution by all PEs
- Providing fourteen 32-bit MCU registers to hold program data and addresses (see section 1.8)
- Transmitting data between the array store or the PE array, and the MCU registers
- Providing hardware support for APAL DO loops (see section 1.9)
- Supporting data transfers between the DAP and the host filestore or attached peripheral devices

1.8 MCU registers and the edge register

The MCU has fourteen 32-bit general-purpose registers that are visible to the low level programmer. There is another register, the *edge* register, whose size matches the size of the array edge (that is, *ES* bits). For example, in a DAP 600, the edge register is 64 bits wide.

Registers can be loaded in various ways from the array store or PE array, or a register's contents can be supplied as data to the memory or PEs. Logical and arithmetic operations are available to operate on the registers. Instructions can test the registers and SKIP on certain conditions, or use them to hold link values for subroutine entry and exit.

You can use a register's contents to modify addresses or values; a register being used in this way is referred to as a modifier register.

In more detail, the functions of the registers are:

- M0 to M13 are general purpose registers, which can hold link values or data, and be operated upon by MCU arithmetic or logical functions. You can also transfer the contents of these registers to or from the array
- M1 to M7 can always be used as modifiers. Register M0 is not generally available as a modifier, since value 0 in the instruction modifier field is interpreted as no modification; the exception is EXIT, where an instruction address rather than a data address is modified. For the DO instruction, you can use registers M1 to M13 as modifiers for the loop count. Instructions J and JE can use M1 to M13 as modifiers
- ME is the edge register, and is matched in size to the array edge size of *ES* bits. In a DAP 500 range machine, ME is the same size as the other registers; in all other ranges of DAP machine, ME is larger than the other registers. The ME register is regarded as being part of the array and can be used as the source or destination of data transferred to or from the array; it cannot in general take part in MCU arithmetic or logical operations or act as a modifier register

Registers M0 to M13 are referred to as MCU registers. You should NOT regard register ME, the edge register, as an MCU register.

In any DAP-series machine, the bits within an MCU register are numbered 0 to 31; within the edge register they are numbered 0 to $ES - 1$. Bit 0 is the most significant bit for both the edge and MCU registers. When data is copied from a register to an array plane, the least significant bit in the register is aligned with the least significant bit in the row or column of the plane, and the register bits transferred across to similarly positioned bits in the row or column. When an MCU register is being used, the remaining ES - 32 bits (if any) in the row or column are zero-filled; when the edge register is used, no zero filling is necessary.

When data is transferred between an array plane and a register you can write the entire ES bits into the edge register, but only the least significant 32 bits into an MCU register (the most significant $ES - 32$ bits, if any, are discarded). Some instructions allow you to address a 32 bit word in the array and to transfer the contents of that word to or from an MCU register (but not to or from the edge register).

1.9 DO loops

A 'DO loop' is a sequence of instructions, invoked by a hardware DO instruction, which is executed a specified number of times, unless a premature exit is taken.

Instructions in a 'DO loop' can, on successive iterations of the loop, access successive bit-planes, rows, columns or words of memory, or bits in an MCU register or the edge register. This process is referred to as 'address stepping'. It is implemented by adding the DO loop iteration number to, or subtracting it from, addresses or values (which can also have the contents of a modifier register added) to give the addresses or values that are 'effective' for that particular execution of the instruction. The result of address generation can thus have 'effective ADDR' (that is, bit-plane number), 'effective INT' (that is, row, column or bit number) and 'effective word address' components. The address generation process is fully described in section 7.1. The iteration number used for address stepping is zero in the first pass of the loop and is incremented at the end of each pass of the loop. Some instruction types can apply this stepping in one of two ways - see section 7.1.2 for more details.

1.10 C- and V-flags

Each of the C-flag (Carry) and V-flag (Overflow) is a 1-bit flag written to by MCU scalar instructions; the flags are normally thought of as holding Boolean values.

The C-flag is affected as follows, the operands being regarded as 32-bit unsigned values:

- For addition, the C-flag is the same as the Carry-out of the most significant bit of the sum
- For subtraction, if operand 2 is less than or equal to operand 1, the C-flag is set to True (that is, 1); otherwise it is set to False (that is, 0). Hence C is the inverse of 'borrow'

Some variants of add and subtract also use the C-flag as the carry-in or inverted borrow-in respectively.

The V-flag is also set according to the result of the above instructions, the operands and result being regarded for this purpose as 32-bit signed (2's complement) integers:

- For addition, if the two operands have the same sign, then the V-flag becomes True if the result sign is different; in all other cases, the V-flag becomes False

- For subtraction, if the two operands have different signs, then the V-flag becomes True if the result sign is different from that of the first operand; in all other cases, the V-flag becomes False

Some of the multiply instructions that return only a single length result set the V-flag according to the discarded most significant half of the product. For further details see the entries for the MPY32V and MPYU32V instructions in appendix F at the back of this manual.

1.11 Array edge dimensions

This manual is general to the whole range of DAP-series machines. The design of APAL is such that you can write source code that is portable across the whole range of DAP-series machines.

The main aspects of the strategy to achieve this edge-size independence are:

- MCU registers are fixed at 32 bits. Thus, when an edge-sized response is returned to the MCU, all but the least significant 32 bits are discarded. Similarly when data is broadcast, the MCU register contents are normally extended on the left (more significant end) with zeros
- The edge register grows in size to match the array dimension
- A DAP word is a 32-bit data item. On DAP 500, words and rows are equivalent; on larger sized arrays, a word is part of a row, and instructions that transfer a single word to or from an MCU register are available. If you want to make sure your DAP code is portable between all edge-sizes of DAP, you should use word addressing rather than row addressing for accessing scalars and addresses held in array store
- Addresses held in modifier registers are always stored in the format of a word address, regardless of whether the modifier is used by a word access instruction or a row access instruction. Thus on the larger edge-size arrays, one or more word-within-row address bits are inserted at the least significant end of the modifier register. In the case of a row access, any word-within-row bits of the modifier are ignored.

  If you want your code to be portable between all edge sizes of DAP, rather than trying to specify address modifiers directly as literals, you should use instructions **RAX** and **RAW** to construct modifier values that are edge size dependent, and so give you the results you want for all edge sizes
- The Assembler permits the size of array for which the code is being assembled to be accessed at assembly time (see section 11.4.2)

Chapter 2

APAL source program format

This chapter covers a number of general points concerned with writing a source program in the Array Processor Assembly Language (APAL).

The characters that can be used in an APAL source program are described in section 2.1.

The format of identifiers in APAL is described in section 2.2.

The format of the various types of data values that can be written in APAL are described in section 2.3.

Continuation lines and comments are described in sections 2.4 and 2.5.

2.1 Character set

2.1.1 Syntax

```
<character> ::= <basic character> | <special character>

<basic character> ::= <letter> | <digit> | [ | ] | . | < | = | > | + | - | / | ( | ) | _ | , |
                      : | & | # | £ | $ | <star> | @ | ; | \ | ' (single quotes) | <space> |
                      <question> | <vertical bar> | { | } | ` (grave) | ~ (tilde)

<special character> ::= ! | ^ | % | "

<letter> ::= A |B |C |D |E |F |G |H |I |J |K |L |M |
             N |O |P |Q |R |S |T |U |V |W |X |Y |Z |
             a |b |c |d |e |f |g |h |i |j |k |l |m |
             n |o |p |q |r |s |t |u |v |w |x |y |z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<alphanumeric character> ::= <letter> | <digit> | '
```

<star> ::= *

<vertical bar> ::= |

<question> ::= ?

2.1.2 Semantics

You can write the source text of an APAL program using any of the characters listed in section 2.1.1. The internal hexadecimal representations of these characters are listed in appendix B at the back of this manual.

You can use a comma as a separator between syntactic constructs in the same way as a space.

For example:

```
FRED: 0 1 3.14 10
```

can also be written as:

```
FRED: 0, 1, 3.14, 10
```

Note, however, that in certain cases commas are obligatory.

The escape character, ^ , can be used to alter the way in which certain special characters are interpreted by the assembler.

The underscore character, _ , can appear anywhere in the source text. If it appears within a character value (see section 2.3.4), it is significant; that is, the assembler treats it like an alphanumeric character. If underscore appears in any other context, it is removed by the assembler. For example, the following are all equivalent:

```
ABCD
AB_CD
AB_C_D
```

whereas the following character values are all distinct:

```
"ABCD"
"AB_CD"
"AB_C_D"
```

Certain APAL instruction mnemonics are commonly written with an underline character to aid clarity, but this is unnecessary; for example the following are equivalent:

```
QT_CF
QTCF
```

Note that the symbols # and £ are equivalent; # is used in this publication.

2.2 Identifiers

2.2.1 Syntax

```
<identifier> ::= <letter> <alphanumeric character>* |
                 '<alphanumeric character>*
```

2.2.2 Semantics

Identifiers are user-defined names that represent instances of various types of entity in APAL. An identifier is a string of up to 32 characters, the first of which must be either a letter or a single quotes symbol. Note however, that by convention only system variables and functions (see sections 10.1 and 11.4.2) begin with a single quote. The single quote, the ' characater, is also known an apostrophe, and has the ASCII value of hexadecimal 27.

The remaining characters in the identifer can be any of the alphanumeric characters described in section 2.1.1. The following are all examples of valid identifiers:

```
NAME
THIS_IS_AN_IDENTIFIER
'PCOUNT
VAR123A
```

Note that two identifiers that differ only in the presence of underscore characters are considered by the assembler to be equivalent.

Identifiers can be used to represent the following APAL entities:

- Modules and module aliases
- Code, data, and mixed sections
- Entry points within code sections
- Data identities
- Data variables
- Code labels
- Assembly-time variables
- Macros, macro parameters, and macro variables

APAL contains no reserved words, although a number of *APAL keywords* have special significance within an APAL program; examples of these keywords include instruction mnemonics and assembly-time statements. APAL keywords are listed in appendix A.

Since APAL keywords are not reserved words you can use them as identifiers, although in the interests of clarity you are recommended not to use them. In particular, you should be aware of the consequences of declaring a macro whose name is the same as an instruction mnemonic or assembly-time statement (see chapter 12).

The assembler considers upper and lower case characters to be equivalent in identifiers and APAL keywords. For example, the following are equivalent:

END_MODULE
Endmodule

2.3 Format of data values

This section describes how the various types of data value that can be manipulated by the assembler are written within an APAL statement.

You can write integer, real, hexadecimal, or character values in an APAL statement. The internal representation of these values is described in chapter 4.

You can also write values that are to be manipulated at assembly time; assembly-time values are described in chapter 11.

2.3.1 Integer values

2.3.1.1 Syntax

<integer value> ::= <unsigned integer> | <signed integer> | <hexadecimal value>

<unsigned integer> ::= <basic integer> | <basic integer>I<basic integer>

<signed integer> ::= <sign>?<unsigned integer>

<basic integer> ::= <digit><digit>*

<sign> ::= + | -

2.3.1.2 Semantics

An integer value, in general, can be:

- An *unsigned integer* value, consisting of a sequence of digits with no intervening spaces, referred to as a *basic integer*
- A *signed integer* value, which is an unsigned integer value that can optionally be preceded by a + or -
- Either form of the above forms of integer value can be followed by an *integer exponent* of the form:

  I *basic integer*

  which causes the preceding signed or unsigned integer value to be multiplied by 10^n, where *n* is the value of *basic integer*.
- A *hexadecimal* value (see section 2.3.3)

Note

+359 is a signed integer; 359 can be considered as either a signed or unsigned integer. If +359 is included in an APAL statement where an unsigned integer is expected, you will get an assembly-time error.

Examples

2349	
-812	
+43I6	(equivalent to 43,000,000; or 43 x 10^6)
-9I2	(equivalent to -900)

2.3.2 Real values

2.3.2.1 Syntax

```
<real value> ::= <sign>?<basic integer><exponent> |
                 <sign>?<basic integer>.<basic integer>?<exponent>? |
                 <sign>?.<basic integer><exponent>?

<exponent> ::= E<sign>?<basic integer>
```

2.3.2.2 Semantics

A real value, in general, can be any of:

- An *unsigned real* value, consisting of two basic integer values separated by a decimal point. Either integer value, but not both, can be omitted
- A *signed real* value, which is an unsigned real value that can optionally be preceded by a + or –
- Either of the above forms of real value can be followed by an exponent of the form:

  E *sign basic integer*

  where *sign* is optional. This causes the preceding signed or unsigned real value to be multiplied by 10^n, where n is the value of *sign basic integer*. If this form is used, then the decimal point can be omitted

Examples

18.	
–.427	
18.427E2	(equivalent to 1842.7)
142E–1	(equivalent to 14.2)
92.E+2	(equivalent to 9200.0)

2.3.3 Hexadecimal values

2.3.3.1 Syntax

<hexadecimal value> ::= #<hexadecimal digit><hexadecimal digit>*

<hexadecimal digit> ::= <digit> | A | B | C | D | E | F | a | b | c | d | e | f

2.3.3.2 Semantics

A hexadeximal value is written as the # character followed by a sequence of hexadecimal digits. A hexadecimal value can be used wherever an integer value can appear.

Examples

#9	(equivalent to 9)
#A	(equivalent to 10)
#1F	(equivalent to 31)

2.3.4 Character values

2.3.4.1 Syntax

<character value> ::= "<value character>*"

<value character> ::= <basic character> | ^<special character>

2.3.4.2 Semantics

A character value is written as a sequence of characters enclosed within double quotes ("). Any of the characters listed in appendix B can be included in a character value, except for the four characters:

! ^ % "

which have special significance for the assembler and are therefore called *signifier* characters. A signifier character can be included in a character value by preceding it with the *escape character* ^ ; the escape character itself is removed by the assembler and does not appear in the value. The escape character can also precede a character that is not a signifier character and will be removed by the assembler, but it cannot be the last character in a character value.

Examples

"ABCD"	(character value ABCD)
"AB^%CD"	(character value AB%CD)
"AB^"CD"	(character value AB"CD)
"AB^ ^CD"	(character value AB^CD)
"AB^ ^ ^%CD"	(character value AB^%CD)

2.4 Continuation lines

Any APAL statement, as defined in section 3.1, can be continued onto any number of succesive *continuation lines*. A line is recognised as a continuation line if it contains a hyphen in the first character position of the line. In the formal syntax of later chapters, the entity <newline> implies the new line character at the end of the last continuation line (if any) of the statement in question.

A source line or continuation line can contain at most 80 characters.

Note that an identifier, a value, a string, or a substitution (see chapter 11) must appear wholly on a single line.

2.5 Comments

2.5.1 Syntax

<comment> ::= ! <comment character>*

<comment character> ::= <basic character> | ^<special character>

2.5.2 Semantics

An APAL program can be documented by the use of *comments*. The start of a comment is indicated by the ! character; a comment is always terminated by the end of the line. A line can consist entirely of comment. A comment can be used in a statement involving continuation lines but the comment is not itself continued, although further comments can be added to the continuation line(s). For example:

```
SVAR = 12.6 +          ! PUTS 12.6 PLUS CONTENTS OF VARIABLES
- SSTART +             ! SSTART
- SEND                 ! AND SEND
                       ! INTO SVAR
```

A comment can include text consisting of any of the characters specified in section 2.1.1.

Note that the ! character is always recognised as signifying a comment, except in the following cases:

- When ! appears within a character value (see section 2.3.4)
- Within a string when it is preceded by ^ (see chapter 11)

Substitutions can take place within comments (see chapter 11).

Chapter 3

APAL source program structure

The APAL assembler processes files of APAL source. These files are first read by a pre-processor, which can insert the contents of other source files where specified by a *file include* statement. Such included files can include other files up to a maximum nesting level of 16. The interface to the assembler and the file include facility are described in *DAP Series: Program Development*; see the version of the manual relevant to your host system.

APAL source text can be interspersed with assembly-time variable definitions or expressions (see chapter 11) and macro definitions or calls (see chapter 12). These definitions are not considered to be part of a module declaration; however, after the effects of such statements have been taken into account, a module declaration must follow the pattern described in chapters 3 to 5. Note that the exact position of an assembly-time variable declaration or macro definition in the source text can affect its scope (see section 3.2).

Each APAL module declaration processed by the assembler results in one CIF module (Consolidator Input Format) as part of a CIF file suitable for input to the DAP linker. The output from the linker is a single DOF file (DAP Object Format) which can be loaded into the DAP code and array stores to form part of the user's DAP program block. Management of CIF files and use of the DAP Consolidator are described in *DAP Series: Program Development*.

This chapter describes:

- How you declare an APAL module
- The contents of an APAL module
- The scope of identifiers within an APAL source program

3.1 Declaring an APAL module

This section describes the declaration and contents of an APAL module.

3.1.1 Syntax

```
<module declaration> ::= <module header><module body>*<module end>

<module header> ::= MODULE<module name><alias>*<newline>

<module name> ::= <identifier>

<alias> ::= <identifier>

<module body> ::= <data section> |              (see chapter 4)
                  <code section> |              (see chapter 5)
                  <mixed section> |             (see chapter 5)
                  <global data identity>        (see chapter 4)

<module end> ::= ENDMODULE <module name>?<newline>
```

3.1.2 Semantics

Each source file you present to the APAL assembler can contain APAL module declarations. An APAL module is the smallest unit of source for which the assembler will generate CIF output.

A module declaration begins with a *module header* of the form:

MODULE *module-name* $alias_1$ $alias_2$ ... $alias_n$

where

module-name is the user-defined name given to the CIF module produced by the assembler for this module

Each of the $alias_i$, which are optional, is an alternative name for the module. The aliases of a module, together with the module name, comprise the module *synonyms*. The synonyms are used by the linker to satisfy outstanding unsatisfied external references (see *DAP Series: Program Development*)

An APAL source module is terminated by:

ENDMODULE *module-name*

where *module-name*, if given, must be the same name as that used in the module header.

You can include in an APAL module any of the following, in any order:

- Data sections, which allocate and optionally initialise areas of the DAP program block and allow these areas to be referred to by name (see chapter 4)
- Code sections, which contain APAL instructions (see chapter 5)
- Mixed sections, which consist of a data part followed by an associated code part (see chapter 5)

- Global data identities, which associate user-defined names at assembly time with addresses in the DAP program block, thereby permitting forward and external references to data (see chapter 4)

Comments can appear anywhere in a module.

3.2 Scope of identifiers

The *scope* of a user-defined identifier is that part of the APAL program to which the identifier is visible; that is, the part in which the name can be referenced in an APAL instruction or assembly-time statement.

The scope of an identifier can be:

- EXTERNAL External identifiers are visible to all modules. An external identifier can be any of the following:
  - A module name or alias
  - Section or entry point names with the HOST or DAP property (see chapters 4 and 5)
  - The name of a macro defined outside a module
  - The name of an assembly-time variable declared outside a module
- GLOBAL Global identifiers are visible throughout the module in which they are declared, but are not visible to any other module. A global identifier can be any of the following:
  - Section or entry point names with neither HOST nor DAP properties (see chapters 4 and 5)
  - The name of a macro defined in the same module
  - The name of an assembly-time variable declared in the same module
  - The name of a data variable declared in a data section in the same module
  - The name on the left hand side of a global data identity (see section 4.3) in the same module
- LOCAL Local identifiers are only visible within the module, or part of the module, in which they are declared (for example, within a particular section). A local identifier can be any of the following:
  - A macro parameter or macro variable (see chapter 12)
  - A code label (see section 5.2)
  - The name of a data variable declared in the data part of a mixed section (see chapter 5)
  - The name on the left hand side of a local data identity

You must declare all identifiers before referencing them, with the following exceptions:

- Code labels
- Code sections and entry points
- Data section names used on the right hand side of global or local data identities

For the purposes of determining the uniqueness of identifiers, the following *classes* of identifiers are distinguished by the assembler:

1 Names of assembly-time variables

2 Names of macros

3 Names of macro parameters and macro variables

4 All other names (see section 2.2)

At any particular point in an APAL program, all the visible identifiers of any one of the above classes must be different. An identifier in one class can be the same as an identifier in another class; for example, a macro can have the same name as an assembly-time variable. Note that the assembler performs no checking of identifiers between modules; the visibility of external identifiers is a linker facility and an error will be flagged at linking time if a clash occurs.

The following table summarises the scope and uniqueness rules described above.

Type of identifier	*Scope*	*Uniqueness*
Assembly-time variable declared outside a module	External	The identifier must differ from all other assembly-time variables
Assembly-time variable declared within a module	Global	The identifier must differ from all other assembly-time variables in the same module and from any assembly-time variables declared previously outside any modules in this assembly
Macro defined outside a module	External	May be the same as any other identifier (see note 1 below)
Macro defined within a module	Global	May be the same as any other identifier (see note 1 below)
Macro parameter	Local	The identifier must differ from all other parameter and variable names in the same macro definition

Type of identifier	*Scope*	*Uniqueness*
Macro variable	Local	The identifier must differ from all other variable and parameter names in the same macro definition
Section or entry point names with HOST or DAP property	External	The identifier must differ from all other class 4 identifiers (as defined on the previous page)
Other section or entry point names	Global	The identifier must differ from all other class 4 identifiers
Data variable declared in a data section	Global	The identifier must differ from all other class 4 identifiers
Data variable declared in data part of mixed section	Local	The identifier must differ from all other class 4 identifiers except local identifiers in other code or mixed sections (see note 2)
Global data identity	Global	The identifier must differ from all other class 4 identifiers
Local data identity	Local	The identifier must differ from all other class 4 identifiers except local identifiers in other code or mixed sections (see note 2)
Code label	Local	The identifier must differ from all other class 4 identifiers except local identifiers in other code or mixed sections (see note 2)
Module name or alias name	External	May be the same as any other identifier

Notes:

1 Only the most recent definition of the identifier is visible. Global identifiers become invisible at the end of the module in which they are declared

2 For the purposes of determining the scope and uniqueness of identifiers the code and data parts of a mixed section are considered to be one section

Chapter 4

Data sections, declarations and identities

An APAL module can contain *data section* declarations each of which reserves part of the array store segment of the DAP program block. A data section contains a number of *data declarations* that define the structure of the data section in terms of the number, type and size of the data values contained in the data section. As part of the data declaration, the data section can either initialise each declared data item, or simply reserve a number of words, rows and planes without initialising the item.

This chapter describes the declaration of data sections and the representation of data values contained in the data sections. Section 4.3 describes *data identities*, which you can use to make forward and external references to data sections.

4.1 Declaring a data section

This section describes the declaration of a data section, which is an area in the array store part of the DAP program block that is to contain data.

4.1.1 Syntax

<data section> ::= <data header><data body>*<data end>

<data header> ::= DATA<data section name><name property>?<common property>?
<write property>?<newline>

<data section name> ::= <identifier>

<name property> ::= DAP | HOST

<common property> ::= COMMON

<write property> ::= WRITE

<data body> ::= <data declaration><newline> | <length><newline>

<length> ::= ROWPACK | WORDPACK

<data end> ::= END<newline>

4.1.2 Semantics

An APAL source module can contain one or more data sections. Each data section begins with a *data header* of the form:

DATA *section-name* *name-property* *common-property* *write-property*

where

section-name is an identifier by which the data section can be referred to in APAL instructions or in data identities (see section 4.3)

name-property, which is optional, specifies to the DAP linker how the data section name is to be linked within the DAP program for the purpose of satisfying external references. *name-property* can take either of the following values:

- DAP The data section is associated with other modules in which *section-name* appears as an external reference. These modules can be derived from source code in either APAL or a high level language such as FORTRAN-PLUS
- HOST As for the DAP property above. The data section can also be referenced in a host program interface subroutine call for the purpose of transmitting data (see the AMT manual *DAP Series: Program Development*). Such a data section must also have the COMMON property (see below)

If you omit *name-property*, the data section is local to the module in which it is declared; that is, an external reference to *section-name* in another module will not be taken as referring to this data section

common-property, which takes the value COMMON, specifies to the DAP linker that the data section is to be mapped onto the same part of the DAP program block as other data sections with both the COMMON property and the same *section-name*. If you omit *name-property*, you must also omit *common-property*.

Any data sections that you map onto such a COMMON area of the DAP program block will begin at the same plane-aligned array store address as other such sections having the same name, although the data sections you map onto the COMMON area need not be of the same size; the size of the COMMON area is the size of the largest data section you have mapped onto that COMMON area.

Only one of the data sections mapped onto a given COMMON area should contain initialisations for its contents, although multiple initialisation will not cause an assembler error *write-property*, which is optional, takes the value WRITE, and specifies that a program has write access to the data section; that is, its contents can be altered by the program. If you omit *write-property*, then the linker places the data section into a 'read only' area of store, unless that section has the COMMON property, and you have specified another data section with the same name in another module in your APAL program, and that other section has COMMON *and* WRITE properties.

Each data item is held in onfi%! Adobe PostScript(tm) via Sun Microsystems PC-NFS % e or more *storag* (32 bits), or one row (a number of bits equal to the array edge size *ES*): in a DAP 500 range machine these two options are identical, but they are different in machines in other ranges. On all ranges by default the storage unit is set to one word (that is, WORDPACK is active) at the start of each data section, but use of the directive ROWPACK changes the size of the storage unit for subsequent declarations. ROWPACK also forces alignment to the next row boundary.

Note that ROWPACK and WORDPACK have no effect on the size of a data item, they merely fix the size of the one or more storage units used to hold each data item.

A data section ends with the line:

```
END
```

The body of a data section consists of data declarations, which are described in the next section.

4.2 Declaring data

This section describes the declarations and optional initialisation of data values within a data section.

4.2.1 Syntax

<data label> ::= <data variable name>:

<data variable name> ::= <identifier>

<data item> ::= <repeat count>?<basic data item> | <repeat count><data sequence>

<data sequence> ::= (<data item><data item>*)

<repeat count> ::= <numval><star>

<numval> ::= <number> | <hexadecimal value>

<number> ::= <unsigned integer> | [<assembly-time expression>]

<value> ::= <integer value> | <real value> | <character value>

<basic data item> ::= <value><size>? | PLANE | ROW | WORD |
PLANE_ALIGN | ROW_ALIGN

<size> ::= (<numval>)

4.2.2 Semantics

A data section can contain data declarations, each of which:

- Optionally labels a word within a data section
- Optionally reserves an area of store within a data section
- Optionally assigns initial values to the reserved area of store. You can specify the number of bits or bytes to be used in the internal representation of values, otherwise default sizes are assumed (see section 4.2.3).

If more than one data section has the COMMON propery, only one of the sections mapped onto the same COMMON area can initialise the area's contents.

The values declared in a data section are mapped onto contiguous areas of array store in the order in which they are declared; the precise nature of the mapping depends on the type of value. A data section is always aligned to a store plane boundary. When assembling a data section, the assembler keeps a record of the within-section address of the next available word, the *data-offset.*

The minimum unit of allocation within a data section is one word or one row, according to the current declared size of the storage unit (that is, according to whether WORDPACK or ROWPACK is active), but the value does not necessarily occupy all the allocated bits (see section 4.2.3).

Hence, when the assembler is mapping the data storage requirements of a DAP program onto the resources available, if the declaration associated with a data item requires 40 bits, and WORDPACK is active, then two storage units are allocated to hold that data item, and *data offset* is increased by 2 words. Of the allocated 64 bits, only the least significant 40 bits will be available to hold that data item, the other bits being zero-filled. This mapping process continues for all data declarations in each data section in the DAP program, with no gap between the storage unit(s) allocated to succeeding data items, unless ROW_ALIGN or PLANE_ALIGN is specified in a declaration (see later).

A data declaration has the general form:

> *data-label: data-item*$_1$ *data-item*$_2$ ... *data-item*$_n$

where

> *data-label* is an identifier, immediately followed by a colon, with global scope within the module in which the data declaration appears. The assembler associates with *data-label* the current value of data-offset, thus allowing the user to refer to that row or word, symbolically rather than by an explicit within-section address (see chapter 7). *data-label* and the following colon, can be omitted, but if present must be the first item in a data declaration
>
> each *data-item*$_i$ specifies an amount of store to be reserved (in multiples of the current storage unit) and optionally provides initial values for that area of store
>
> each *data-item*$_i$ can be either a *basic data item* or a *data sequence*.

4.2.2.1 A basic data item

A basic data item can be any of the following:

- WORD The next word of store is reserved, with no initial value. Data-offset is increased by one word. The directive WORD is not allowed when the current storage unit is a row (that is, when ROWPACK is active)
- ROW The next row of store is reserved, with no initial value. Data-offset is increased by one row. For machines with an array edge size *ES* greater than 32, which have more than one word per row, if data offset is not at a row boundary when ROW is used, data offset is increased to the start of the next row boundary before the row is allocated
- PLANE The next store plane is reserved, with no initial value. If data-offset is not at a plane boundary when PLANE is used, data-offset is increased to the start of the next plane boundary (as with PLANE_ALIGN), then in addition a complete store plane is reserved (unlike PLANE_ALIGN)
- ROW_ALIGN If data-offset is not aligned to a row boundary (only possible on machine with an *ES* greater than 32), the offset is increased to the next row boundary. This makes sure that the next item to be declared in that section is aligned to the first word of the next available row
- PLANE_ALIGN If data-offset is not aligned at a plane boundary, it is increased to the next plane boundary. This makes sure that the next item to be declared in that section is aligned to the first storage unit of the next available store plane
- *value* (*size*)

  where *value* is an integer, real, hexadecimal, or character value as defined in section 2.3. *size*, which is optional (along with its enclosing brackets), is an unsigned integer or hexadecimal value, or an assembly-time expression, in [ ], yielding a positive integer value (see chapter 11). *value* is allocated to the next available storage unit(s), starting at the word defined by data-offset, and data-offset is incremented accordingly. For example:

  ```
  3.4(32)
  ```

  reserves space and initialises the space to a 32-bit real value

The internal representations of values and how they are mapped onto the array store is described in section 4.2.3.

Note that a minus sign (hyphen) in the first column of a source line indicates a continuation line rather than a negative value.

A basic data item can optionally be preceded by the construct:

*count**

where *count* is an unsigned integer or hexadecimal value, or an assembly-time expression, in [ ], yielding a positive integer value. The effect of this construct is as though *n* consecutive instances of the basic data item had been written, where *n* is the value of count.

For example, the following declarations are equivalent:

```
VAR1 : 12.6   12.6  PLANE   PLANE   14
VAR1 : 2*12.6   2*PLANE   14
```

Note that on all DAP-series machines, PLANE is equivalent to:

PLANE_ALIGN *ES**ROW

where *ES* is the edge size of the machine concerned,

and that *count**PLANE_ALIGN has the same effect as PLANE_ALIGN, but if you try to use *count**PLANE_ALIGN the assembler will output a warning message.

A *count* of zero is an error.

4.2.2.2 A data sequence

A data sequence has the form:

*count**(*data-item*$_1$ *data-item*$_2$ ... *data-item*$_n$)

where each *data-item*$_i$ can be a basic data item or another data sequence. The characteristic of a data sequence is the use of the enclosing '( )' and of *count**; a sequence allows you to declare one or more instances of a data structure.

For example:

```
STRUCT1: 3 * (1,2,2 * (3,4),5)
```

is equivalent to the declaration:

```
STRUCT1: 1  2  3  4  3  4  5
         1  2  3  4  3  4  5
         1  2  3  4  3  4  5
```

Data-offset is incremented for each basic data item in the data sequence.

If you want to use either or both of PLANE_ALIGN or PLANE in a data sequence, the first basic data item in the outermost data sequence must be PLANE_ALIGN; you can use PLANE_ALIGN and PLANE freely after that.

Likewise, if you want to use either or both of ROW or ROW_ALIGN in a data sequence, the first basic data item in the outermost data sequence must be either ROW_ALIGN or PLANE_ALIGN; you can use ROW and ROW_ALIGN freely after that. This requirement to start with ROW_ALIGN or PLANE_ALIGN is not necessary with machines in the DAP 500 range, as all data items are row aligned by default.

4.2.2.3 Mapping of values onto array store

You should be aware of the way in which the assembler maps values onto the array store; note the following points:

- If a data label is immediately followed by PLANE_ALIGN, then the label refers to the value of data-offset before PLANE_ALIGN is implemented rather than after.

  For example, if ROWPACK is in force (that is, the size of a storage unit is a row):

  - LABEL1: PLANE_ALIGN 3 6 9

    If the current value of data-offset just before the above declaration is, say, plane 6 row 7, LABEL1 will refer to this row, whereas the value 3 will be stored in plane 7 row 0.
  - LABEL2: 3*(PLANE_ALIGN 1 PLANE 2)

    If data-offset is plane 4 row 7 just before the above declaration, the assembler performs the following mappings:

    1. LABEL2 is associated with plane 4 row 7
    2. Data-offset is incremented to plane 5 row 0
    3. The value 1 is stored in plane 5 row 0 and data-offset is incremented by one
    4. Data-offset is incremented to plane 6 row 0 and an entire plane is allocated; data-offset is now plane 7 row 0
    5. The value 2 is stored in plane 7 row 0 and data-offset is now plane 7 row 1
    6. The equivalents of steps 2 to 5 are repeated twice more to give the following final mapping, where ES is the edge size of the DAP-series machine concerned:

Plane	*Row*	*Value*
4	7	Labelled by LABEL2; no initial value
	8 to $(ES-1)$	No initial value
5	0	Value 1
	1 to $(ES-1)$	No initial value
6		No initial value
7	0	Value 2
	1 to $(ES-1)$	No initial value
8	0	Value 1
	1 to $(ES-1)$	No initial value
9		No initial value
10	0	Value 2
	1 to $(ES-1)$	No initial value
11	0	Value 1
	1 to $(ES-1)$	No initial value
12		No initial value
13	0	Value 2

- If a nested data sequence (that is, a data sequence that is an item in another data sequence) contains either PLANE or PLANE_ALIGN, then unless the nested data sequence begins with PLANE_ALIGN each instance of the nested data sequence need not start at the same row within a plane.

  Similarly, for machies with *ES* greater than 32, if a nested data sequence contains either ROW or ROW_ALIGN, unless the nested data sequence begins with ROW_ALIGN each instance of the nested data sequence need not start at the same word within a row.

  For example, if ROWPACK is active:

  ```
  PLANE_ALIGN
  LABEL: 2 * (PLANE_ALIGN 1 2* (1,2,3 PLANE) 4 5)
  ```

  produces the following mapping (assuming a current data-offset of plane 8 row 0, and that *ES* is the machine edge size):

Plane	*Row*	*Value*
8	0 to 3	Values 1, 1, 2 and 3
	4 to $(ES - 1)$	No initial value
9		No initial value
10	0 to 2	Values 1, 2 and 3
	3 to $(ES - 1)$	No initial value
11		No initial value
12	0 and 1	Values 4 and 5
	2 to $(ES - 1)$	No initial value
13	0 to 3	Values 1, 1, 2 and 3
	4 to $(ES - 1)$	No initial value
14		No initial value
15	0 to 2	Values 1, 2 and 3
	3 to $(ES - 1)$	No initial value
16		No initial value
17	0 and 1	Values 4 and 5

Note that the first and third instances of the inner data sequence begin at row 1 of a plane, whereas the second and fourth instances begin at row zero

When you are declaring complex data structures make sure that data sequences containing PLANE_ALIGN or PLANE are plane aligned, and that data sequences containing ROW_ALIGN or ROW are either row aligned or plane aligned.

4.2.3 Representation of values in the array store

This section describes how values specified in data declarations are represented within the array store.

Data values are held in the machine as bit patterns that are contained in one or more words or rows. APAL provides four different types of data value: integer, real, hexadecimal and character.

The storage allocated to each type of value depends on the data declaration for the data item concerned, with a maximum of 64 bits for integer and real values. For hexadecimal values, a maximum allocation is either 64 bits or *ES* bits, whichever is the greater; for character values, the maximum is 512 bits. The three numeric types differ only in the notation used to represent the bit patterns; they do not reflect any intrinsic properties of the bit patterns in store. For example, a bit pattern of 32 zero bits could be written as any of:

```
0(32)            (integer)
0.0(32)          (real)
#00000000        (hexadecimal)
```

with the same effect in each case.

4.2.3.1 Integer values

An integer value is represented exactly. When you specify an integer value in a data declaration you can specify the number of bits that are to be used to represent the value by using the (*size*) option. For example:

```
VARINT : 12I7(29)          ! INTEGER VALUE IS REPRESENTED IN 29 BITS
```

The maximum value of *size* is 64 bits, the default value of *size* is 32 bits.

An integer value is held, in two's complement form, in the rightmost (least significant) n bits of the storage unit(s) needed to contain the n bits, where n is the value (possibly default value) of *size*. Any remaining bits of the storage unit(s) are set to zero.

The range of integer values that can be held in this way depends on the value of *size*. An n-bit integer can take any value in the range -2^{n-1} to $+2^{n-1} - 1$; if you try to enter a value outside this range the assembler will flag an error.

4.2.3.2 Real values

In general a real value is held as an approximation to the specified real value. When you specify a real value in a data declaration you can specify the number of bits that are to be used to represent the value by using the (*size*) option.

For example:

```
    VARREAL : 916.7234617(40)
```

For a real value *size* can be 24, 32, 40, 48, 56 or 64 bits; if (*size*) is omitted, (32) is assumed as default.

A real value is held in FORTRAN-PLUS floating point format in the rightmost n bits of the storage unit(s) needed to contain the n bits, where n is the value (possibly default value) of *size*. That is:

- The leading *tot_bits* − n bits are set to zero, where *tot_bits* is the number of bits in the storage unit(s) allocated by the relevant data declaration
- The rightmost $n - 8$ bits represent the mantissa as an unsigned binary fraction. If the value being represented is zero, then all those $n - 8$ bits are zero. For a non-zero value, the mantissa is normalised so that it is a fraction in the range

  $1/16 \leq \text{mantissa} < 1$

- The remaining eight bits of the representation are used to hold the sign bit (zero for a positive value and one for a negative value) followed by a seven-bit exponent, which is an unsigned integer. The value so represented is:

  $(-1)^{sign} \times 16^{(exponent-64)} \times mantissa$

  The range of values that can be represented in this way is +/− (5.4×10^{-79} to 7.2×10^{75}) approximately. Values in the range -5.4×10^{-79} to 5.4×10^{-79} approximately, are represented as zero.

  The precision with which a real value is represented depends on the number of bits in the mantissa, and therefore depends on the value of *size*. The following table shows the relationship between *size* and precision:

Size	*Precision in significant decimal digits (approximately)*
24	4
32	7
40	9
48	11
56	14
64	16

4.2.3.3 Hexadecimal values

A hexadecimal value is represented exactly; it must be a positive value. When you specify a hexadecimal value in a data declaration you can specify the number of bits that are to be used to represent the value by using the (*size*) option. For example:

```
    VARHEX1 : #123F(24)          ! 24 BITS ARE USED
    VARHEX2 : #1A3               ! 32 BITS ARE USED
```

The maximum value of *size* is 64 bits or *ES* bits, whichever is the greater; the default value of *size* is 32 bits.

A hexadecimal value is held in the rightmost (least significant) n bits of the storage unit(s) needed to contain the n bits, where n is the value (possibly default value) of *size*. Any remaining bits of the storage unit(s) are set to zero.

The range of hexadecimal values that can be held in this way depends on the value of *size*. An n-bit hexadecimal value can represent an unsigned integer in the range zero to 2^n - 1; if you try to enter a value outside this range the assembler will flag an error.

4.2.3.4 Character values

A character value is held in a number of consecutive storage units, with each character occupying 8 bits. If necessary, the assembler adds space characters to the left to make the storage allocated to the value a whole number of storage units.

When you specify a character value in a data declaration you can specify the number of characters in the value to be allowed for, by using the (*size*) option to specify the number of bits to be allocated. For example:

```
WORDPACK
VARCHAR : "EXAMPLE CHARACTER VALUE"(192)       ! 24 CHARACTERS
                                               ! (6 STORE WORDS)
                                               ! ARE USED
```

If (*size*) is omitted, a default number of bits of eight times the number of characters in the value is taken. Character values are padded, if necessary, with leading space characters.

The above example character value would be stored in consecutive words as follows, starting at the store address labelled as VARCHAR:

```
 EXA
MPLE
 CHA
RACT
ER V
ALUE
```

The maximum size for a character value, whether a declared *size* or a default value, is 64 characters, 512 bits. As usual, if you try to insert into the requested storage space a value too large to fit, you will get an assembly-time error. Each character in a character value is represented by an eight-bit ASCII-like pattern, as specified in appendix B.

4.3 Data identities

A data identity associates, at assembly time, an identifier with an array store address, and is similar to a *data label*, already discussed above. Once declared, an identifier can then appear in subsequent APAL statements, where it will be replaced by the address assigned to it in the data identity. Note that a data identity does not reserve storage; it is merely an assembly-time association between an identifier and a store address. Data identities are the only way you have of getting forward or external references to the contents of a data section.

4.3.1 Syntax

<data identity> ::= <global data identity> | <local data identity>

<global data identity> ::= DEFINE<newline><identity>*END<newline>

<local data identity> ::= <identity>

<identity> ::= <identity name> = <data address><newline>

<identity name> ::= <identifier>

<data address> ::= <plane> | <row> | <word> | <column>

<plane> ::= <aligned data name><plane offset>? | <plane number>

<name or plane> ::= <data name><plane offset>? | <plane number>

<row> ::= <name or plane><row offset>? | <row offset>

<column> ::= <name or plane><column offset>? | <column offset>

```
<word> ::= <row> |
           <name or plane>?.<word offset> |
           <name or plane>?<row offset><word offset>
```

<aligned data name> ::= <data name>

<data name> ::= <data section name> | <data variable name> | <identity name>

<plane offset> ::= + <plane number>

<row offset> ::= .<numval>

<column offset> ::= .<numval>

<word offset> ::= .<numval>

<code store address> ::= <within-section address> | <inter-section address>

```
<within-section address> ::= <code label name><label offset>? |
                             <star><label offset><doj modifier>?
```

```
<inter-section address> ::= <code section name><section offset>?<doj modifier>? |
                            <entry point name><section offset>?<doj modifier>?
```

<label offset> ::= +<numval> | −<numval>

<section offset> ::= +<numval>

4.3.2 Semantics

Generally, you should not reference the contents of a data section in APAL statements until you have declared the data section. However, you can get forward references to data sections in the same module, and external references to data sections in other modules, by using data identities, which associate an identifier with an array store address at assembly time. Data identities can refer to previously declared data sections, data variables and data identities. You can use data identities to associate names with locations in the workspace part of the DAP program block (see section 1.4 for further details), in which the plane addresses are in the range 0 to 119.

There are two classes of data identity:

- Global data identities. These can appear within an APAL module declaration. They cannot appear in a data, code, or mixed section, and must be enclosed within the keywords DEFINE and END:

  ```
  DEFINE
  .
  identities
  .
  .
  END
  ```

  The identifiers appearing on the left hand side of global data identities have global scope within the module in which they are defined

- Local data identities. These can appear in a code section, or in the code part of a mixed section. The keywords DEFINE and END must not appear. The identifiers that are defined in local data identities have local scope within the code section in which they are defined

A data identity has the form:

> *identity-name = data address*

where

> *identity-name* is the name that can appear in subsequent APAL statements. *identity-name* must be different from all other class 4 names currently in scope (class 4 names are defined in section 3.2)
>
> *data address* specifies the array store address that is to be associated at assembly time with *identity-name*. *data address* can have any of the forms:
>
> - *data name plane-offset row-offset word-offset*
>
>   where
>
>   *data-name* can be the name of a data section in the same or another module, a data label in a previously declared data section, or the name of a previously defined data identity
>
>   *plane-offset*, *row-offset* and *word-offset*, which are all optional, have the form:
>
>   *+numval .numval .numval*
>
>   where *numval* is an unsigned integer or hexadecimal value, or an assembly-time expression within [ ] yielding a non-negative integer value

This form of *data address* specifies an address relative to the address of a previously defined data name or an as yet undeclared data section.

For example:

FRED = TOM + 12.6

associates with FRED the store address 12 planes and 6 rows beyond TOM which can be the name of a preceding or subsequent data section, a data label in a previously declared data section, or a previously defined data identity;

MARY = FRED + 2.1.1

associates with MARY the store address 2 planes, 1 row and 1 word beyond FRED, that is 14 planes, 7 rows and 1 word beyond TOM, which in a DAP 500 would be 14 planes and 8 rows beyond TOM;

FRIEDA = MARY + 1..1

associates with FRIEDA the store address 1 plane and 1 word beyond MARY, or 3 planes, 1 row and 2 words beyond FRED (3 planes and 3 rows beyond FRED on a DAP 500, 3 planes and 2 rows beyond FRED on a DAP 600), and so on

- *plane-number row-offset word-offset*
  where

  plane-number has the form:

  numval

  where *numval* is an unsigned integer or hexadecimal value, or an assembly-time expression within [ ] yielding a non-negative integer value

  row-offset and *word-offset*, which are optional, are as defined on the previous page

This form of *data address* specifies an address that can be regarded, depending on the context in which it is used, as either:

- An address relative to the start of the array store part of the DAP program block
- An address relative to an address held in a modifier register

Consequently the name is associated with any or all of a plane, row or word displacement without specifying the base for this displacement. For example, given the data identity:

FRED = 27.16.1

the instruction:

RW M3 FRED

loads the contents of the word at location plane 27 row 16 word 1 into M3. However, the instruction:

RW M3 FRED(M6)

loads M3 with the contents of the word 27 planes, 16 rows and 1 word beyond the address held in M6

- *row-offset word-offset*

  where *row-offset* and *word-offset* are as defined for the first and second forms of *data address*. Either or both of *row-offset* and *word-offset* must be present in this form of *data address*.

  This form of data address specifies either or both of a row or word displacement; for example:

  ```
      FRED = .78.1
  ```

  associates with FRED the address 78 rows and 1 word beyond an address in a modifier register. Alternatively (depending on the context), FRED = .78.1 could associate FRED with a particular word in a particular plane, the actual word depending on the particular range of DAP machine concerned. On a DAP 500 machine this address would be plane 2 row 15; on DAP 600 it would be plane 1 row 14 word 1

As you will have seen from the examples above, in all the three forms of data address for all ranges of DAP-series machines, you can use a word-offset equal to or greater than the number of words in a row, in which case the word-offset count is carried across row boundaries. Similarly, you can use row-offset equal to or greater than the number of rows in a plane, and the row-offset count is carried across store plane boundaries. Hence, the following data identities are all equivalent in a DAP 600-series machine:

```
JANE = 10.152.4
JANE = 11.89.2
JANE = 12.26
```

Example global data identity on a DAP 600 machine

Assuming that a data section called DATAONE is declared elsewhere, the following global data identity can appear in a module declaration:

```
DEFINE

FIRST = DATAONE + 12.3      ! ASSOCIATES 'FIRST' WITH AN ADDRESS 12 PLANES
                            ! AND 3 ROWS BEYOND THE START OF THE DATA SECTION
                            ! CALLED (OR YET TO BE CALLED) 'DATAONE'

SECOND = 14.2.1             ! ASSOCIATES 'SECOND' WITH AN ADDRESS OF PLANE
                            ! 14 ROW 2 WORD 1 IN THE ARRAY STORE PART OF
                            ! THE DAP PROGRAM BLOCK

THIRD = DATAONE.30          ! ASSOCIATES 'THIRD' WITH AN ADDRESS 0 PLANES AND
                            ! 30 ROWS BEYOND THE START OF DATAONE

FOURTH = THIRD + 22.1.1     ! ASSOCIATES 'FOURTH' WITH AN ADDRESS 22 PLANES
                            ! 31 ROWS AND 1 WORD BEYOND THE START OF DATAONE

FIFTH = FOURTH + 5..3       ! ASSOCIATES 'FIFTH' WITH AN ADDRESS 27 PLANES
                            ! AND 33 ROWS BEYOND THE START OF DATAONE

END
```

Chapter 5

Code sections

An APAL module can contain *code section* declarations, each of which can contain any number of the following in any order:

- Entry point declarations (see section 5.2.1)
- APAL instructions (see section 5.2.2)
- Local data identities (see section 4.3)
- The TRACE statement (see chapter 8)

An APAL module can also contain any number of *mixed section* declarations, each of which consists of a code section declaration preceded by a data section whose contents are local to that code section.

The declaration of a code section is described in section 5.1. The contents of a code section are described in section 5.2. The declaration and contents of a mixed section are described in section 5.3.

5.1 Declaring a code section

5.1.1 Syntax

<code section> ::= <code header><code body>*<code end>

<code header> ::= CODE<code section name><name property>?<newline>

<code section name> ::= <identifier>

<name property> ::= DAP | HOST

```
<code body> ::= <entry point> |
                <code label>?<APAL instruction>?<newline> |
                <local data identity> |
                <TRACE statement>

<code label> ::= <code label name>?:

<code label name> ::= <identifier>

<code end> ::= END<newline>
```

5.1.2 Semantics

An APAL source module can contain code sections. Each code section begins with a *code header* of the form:

CODE *section-name* *name property*

where

section-name is an identifier by which the section can be referenced in instructions within the section and by instructions in other code sections. The scope of *section-name* can be global or external, depending on the value of *name-property* (see below)

name-property, which is optional, specifies to the DAP linker how the code section is to be linked within the program; it can take either of the following values:

- DAP The code section can only be linked within the DAP program; that is the code section can only be entered from another APAL code section or from a FORTRAN-PLUS program
- HOST The code section can be linked within the DAP program (as with the DAP property); the section can also be referenced in a host program interface subroutine call for the purpose of entering the DAP program (see the AMT manual *DAP Series: Developing and using DAP Programs*).

If *name-property* is omitted, the code section can only be entered from a code section declared in the same module; *section-name* is an identifier with external scope if *name-property* is specified, otherwise it has global scope

A code section ends with the line:

END

5.2 Code section contents

5.2.1 Declaring entry points

This section describes the syntax of entry point declarations and their function.

5.2.1.1 Syntax

<entry point> ::= ENTRY<entry point name><name property>?<newline>

<entry point name> ::= <identifier>

5.2.1.2 Semantics

A code section is normally entered at the first APAL instruction in the code section. However, any number of alternative entry points can be defined by writing:

ENTRY *entry-point-name* *name-property*

at the required point in the code section, where *entry-point-name* and *name-property* have the same functions as in a code header (see section 5.1). Note, however, that the *name-property* specified in an entry point declaration need not be the same as the *name-property* specified in the code header.

When a code section is entered via an entry point, the first instruction to be executed is the first instruction following the entry point declaration.

5.2.2 APAL instructions

The APAL instructions in a code section specify the operations to be performed when that code section is entered.

Any APAL instruction can be preceded by a code label, which has the form:

label:

where *label* is an identifier with local scope in the code section; there must not be any spaces between the identifier and the colon. The code label can be used by instructions in the same code section to transfer control to the instruction that it labels.

A code label can appear without an APAL instruction following it, in which case it refers to the next APAL instruction for which the assembler generates object code.

A *dummy* code label, consisting of a colon with no preceding label identifier can be used to identify the last instruction in an APAL DO loop. A dummy code label has no other significance.

APAL instructions are described in appendix F to this manual.

5.3 Mixed sections

This section describes the declaration and contents of a mixed section.

5.3.1 Syntax

<mixed section> ::= <mixed header><data body>*<code section>

<mixed header> ::= MIXED<data section name><name property>?<common property>?
<write property>?<newline>

5.3.2 Semantics

An APAL module can contain *mixed sections*. Each mixed section consists of a code section preceded by a data section whose contents are local to that code section.

A mixed section begins with a *mixed header* of the form:

MIXED *data-section-name name-property common-property write-property*

where *data-section-name*, *name-property*, *common-property* and *write-property* refer to the data section part of the mixed section and have the same functions as when applied to a data section (see section 4.1).

The mixed header is followed by data declarations, which are as described in section 4.2 except that data labels are local to the mixed section rather than global within the whole module and must be different from all external and global identifiers in the module (see section 3.2).

The last data declaration is followed, without an END line, by a code section header, as defined in section 5.1. The code section name, and the name property (if specified), refer to the code part of the mixed section and have the same functions as when applied to a code section. Note that the code and data sections associated with a mixed section must have different names.

The code section name can be followed by any number of code section statements; the line:

END

must terminate the mixed section.

For the purposes of linking by the DAP Consolidator, as specified by the properties in the mixed and code section headers, the data and code sections of a mixed section are treated as separate sections. However, for the purpose of determining the scope of identifiers within the mixed section the data and code sections are regarded as a single section; that is, all identifiers declared within the mixed section have local scope within that section.

Chapter 6

APAL Instructions

This chapter describes the fields that an APAL instruction can contain, and gives an introduction to the detailed description of individual instructions you will find in appendix F at the end of the manual. This chapter (and chapter 7) is not intended to give you detailed knowledge and understanding of the bit patterns of different APAL instructions. It is more concerned with giving you a general understanding of the functions of the various fields in an APAL instruction.

Section 6.1 is concerned with the hardware instruction set in the DAP. In appendix F you will find details of the hardware instructions, together with those of 'pseudo' instructions. Pseudo instructions are 'translated' by the assembler into one of several hardware instructions, the actual hardware instruction depending on the value of certain fields in the pseudo instruction. For example, you can use the RLIT pseudo instruction to load a literal value into an MCU register. If all but the 16 least significant bits in the literal are zeros, then the assembler generates an RH instruction, which copies the literal into the register specified in RLIT. In some other cases, the assembler creates the literal in array store, then generates an RX instruction to load that literal into the specified register.

APAL instructions can be split up into two broad categories: MCU instructions, in which only the MCU is involved; and array instructions, in which the array of PEs is also involved. Section 6.2 breaks the instruction set down further – into 17 categories – lists the instructions in each category, and gives a 1-line description of each instruction. The section also discusses execution times for the different instructions.

6.1 Instruction fields

This section describes the fields that APAL hardware instructions can contain, and gives the associated syntax where this is not covered elsewhere.

Each APAL instruction is assembled into a 32-bit instruction word; these instructions occupy consecutive words in the code store when the DAP program is loaded.

Some patterns of bits in the instruction word do not correspond to a valid instruction. The assembly system generally prevents the creation of instructions having such bit patterns, but in the unlikely event of one being encountered when the program is executed, a run-time error results.

Each instruction includes an OPERATION field which, broadly, is the op-code for the instruction. As with conventional microprocessor chips, the field is used by the MCU to decode the remaining bit pattern in the instruction and to interpret that pattern as one or more instruction fields, essentially containing parameters to the op-code. Unlike conventional microprocessors, the APAL instruction has a fixed 32-bit length. All the various possible fields that can occur in an instruction are discussed in the following sub-sections; no instruction will contain all the fields.

6.1.1 The OPERATION field

The OPERATION field specifies the essential function of the instruction; more detailed information on the instruction is given in the rest of the instruction word. The OPERATION field specifies whether registers are to be written, whether data is to be broadcast to the PE array, which function is to be performed by the scalar Arithmetic and Logic Unit (ALU) within the MCU and the form of jumps and DO loops. For array instructions, the field also specifies the detailed function of the array, such as whether array store is to be written, whether PE registers are to be clocked, which operands are to be input to the PE, whether an entire store plane or just one row, column or word is to be used, and whether a response is to be returned to the MCU.

No further detail of the OPERATION field is given here, since its content is implicit in the descriptions of individual instructions given in appendix F.

6.1.2 The MCUR field

The MCUR field is a field that identifies an MCU register or the edge register that is to be used as either or both of the source and destination of data. Any of MCU registers M0 to M13 can be used for this purpose; for many instructions, the edge register can be used instead.

The register identified by the MCUR field can be one of:

- An MCU register used as the first operand of a scalar arithmetic, logical or shift instruction. In this case the register specified in this field will also contain the result of the instruction. The edge register can only be specified for a subset of scalar shift operations
- An MCU register or the edge register whose contents are broadcast to the array
- An MCU register or the edge register from which a single bit is selected for testing by the MCU or for broadcast to the array
- An MCU register or the edge register to which a response from the array is to be written

The instruction syntax is one of:

```
<MCU-register> ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7 | M8 | M9 | M10 | M11 |
                   M12 | M13 |
<MCU-or-edge-register> ::= <MCU register> | ME
```

Which syntax is used depends on the actual instruction concerned.

6.1.3 The MOD field

The MOD field specifies a second MCU register, or in a few cases, the edge register. The use of the register varies with the type of instruction:

- For array instructions, the register contributes to address generation, as described in sections 7.1.1, 7.1.2, 7.1.3, or 7.1.4. Any of registers M1 to M7 can be specified; if none is specified, a value to signify no address modification is encoded in the MOD field
- For MCU instructions DO, J, JE and EXIT the register contributes to address generation as described in sections 7.1.5, or 7.1.6. Any of registers M1 to M13 can be specified; if none is specified, a value to signify no address modification is encoded in the MOD field
- For scalar arithmetic, logical or shift instructions, the register holds the second (or only) operand. Any of registers M0 to M13 can be specified. The edge register can only be specified for a subset of scalar shift operations.

The syntax of modifiers is given in section 7.3.1.

6.1.4 The ADDR, INT and WORD fields

The ADDR field specifies a bit-plane address within array store; the field is eight bits long.

Depending on the instruction, the INT field can specify a row or column number in array store; a bit number within an MCU register or the edge register; or, for an MCU register shift, array shift or vector add instruction, the count. The length of the field depends on which range of DAP-series machine the APAL code is assembled to run on. For DAP 500 machines, the INT field is 5 bits long; for the DAP 600 range, the field is 6 bits.

The WORD field allows you to specify the word component of an array store address. As with the INT field, the length of the WORD field depends on the machine the APAL code is assembled to run on. WORD is 0 or 1 bits long, for the DAP 500 and 600 ranges of machine respectively; that is, the WORD field does not exist on code assembled to run on a DAP 500.

The use of these three fields in address generation is discussed in section 7.1, and the syntax of addresses in section 7.3.

6.1.5 The INCREMENT/DECREMENT and STEP TYPE fields

The INCREMENT/DECREMENT field is only relevant if the instruction containing it appears inside an APAL DO loop; the field specifies how addresses in the instruction are to be stepped within the DO loop. The three options are:

- Address is unaffected
- Address is to be incremented
- Address is to be decremented

For some instructions there is an option to step the bit-plane address or the row or column address; the option is specified in the STEP TYPE field.

The syntax of DO loop stepping is given in section 7.3.2

6.1.6 The DIRECTION and GEOMETRY fields

The DIRECTION field specifies the direction of shift for an array shift operation, or the direction of carry propogation for an array vector add operation.

For such instructions, the edge effects (or topology) are specified by the GEOMETRY field. The available geometries are 'Plane', meaning that zero (or False) is input at the edge, or 'Cyclic', meaning that data shifted out at one edge is shifted in at the corresponding row or column position at the opposite edge.

The options available with the DIRECTION and GEOMETRY fields are described in section 7.1.4.

6.1.7 The LITERAL field

The LITERAL field is used as the second operand of some scalar arithmetic and logical instructions. The field is 16 bits wide, and before use in the scalar operation it is implicitly expanded to a 32 bit value by extending on the left with zeros – that is, zeros are inserted at the most significant end of the field.

The syntax of such a literal is:

```
<literal-16> ::= <integer value><size>? | <character value><size>?
<size> ::= (<numval>)
```

Thus the literal can be any of:

- An optionally signed integer, optionally followed by (size) in units of bits, for example -24(6). The maximum value of 'size' is 16; if (size) is omitted, its default value is (16). 'size' represents the number of right-aligned bits of the field used to hold the integer, which is represented in two's complement form. The remaining bits of the 16-bit field are set to zero. If size = n, the range of integers that can be so represented is -2^{n-1} to $2^{n-1}-1$
- A hexadecimal value, followed by (size) in units of bits, for example #3E(10). The maximum value of 'size' is 16; if 'size' is omitted, its default value is (16). The value is held right-aligned in the instruction with the remaining bits set to zero. The maximum hexadecimal value is #FFFF
- A character value of zero, one or two characters, followed by (size) in units of bits, for example "J"(8); the maximum value of 'size' is 16. If 'size' is omitted, its default value is eight times the number of characters given, with a maximum of 16. The value is held right-aligned, and is padded on the left with a space character if necessary

6.1.8 The CODE ADDRESS field

The CODE ADDRESS field is used by instructions that jump to another instruction. The field can specify the address of an instruction anywhere in the DAP program block.

The syntax of code store addresses is given in section 7.3.5.

6.2 The instruction set

6.2.1 Introduction

DAP hardware instructions are classed in one of 16 'groups', depending on the general type of function performed. Which group a particular instruction belongs to is defined in part of the OPERATION field of the instruction. This section discusses the instruction set, listing the hardware instructions in each group, and giving a 1-line description for each instruction. The section also covers the pseudo instructions.

The hardware instruction set includes some *compound instructions*: in these instructions, the actions of two component instructions are executed as a result of the one compound instruction, whose mnemonic is constructed by adding together the mnemonics of the two component instructions; by convention, the component mnemonics are separated by the '_' character. For example, the instruction **AS_CF** copies a specified store plane into the PE A registers (instruction **AS**), then sets every bit of the C plane to zero (instruction **CF**). From a functional point of view, the component instructions are always carried out in the order they appear in the compound instruction.

There is also a compact list of mnemonics in appendix A; tables giving the derivation of the mnemonics are presented in appendix E.

In the descriptions below, as elsewhere, the complete set of Q, C or A single-bit registers for all the PEs is referred to as the Q, C or A plane respectively. Also, *inverting* a plane means inverting each bit in the plane. Thus 'Q plane = C plane + store plane' means that the Q register bit, in every PE, is to be set equal to the sum of the C register bit and the bit 'belonging' to that PE in the plane whose address is specified in the instruction. Although not explicitly specified in the entries below, any carry bits from such sums are discarded. See appendix F for more details of any particular instruction. For compactness the entries below present each instruction on a single line.

Some instructions (groups 6 and 7) broadcast a register to the PE array or to a store plane. The options available depend on the size of the target DAP machine, and on whether an MCU register or the edge register is to be broadcast, as explained in section 1.8 and 1.11. In the following subsections and in appendix F, the term *R-plane* is used to refer to the notional or fictitious plane resulting from that broadcasting, where every row in the plane is identical; the term *orthogonal R-plane* is applied to the notional plane where every column is identical. There is no physical R-plane or orthogonal R-plane, they are simply concepts used in describing the operation of certain instructions.

6.2.2 Group 0

General characteristic: instructions fetch an array store plane and combine it with PE register plane(s).

AS(N)	A plane = (inverted) store plane
AMS(N)	A plane = A plane ANDed with (inverted) store plane
AS(N)_CF	A plane = (inverted) store plane, C plane = zeros
QS(N)	Q plane = (inverted) store plane
QS_AS(N)	Q and A planes = (inverted) store plane
QPQS(N)	Q plane = Q plane + (inverted) store plane
QPCQS(N)	Q plane = C plane + Q plane + (inverted) store plane
QPCS(N)	Q plane = C plane + (inverted) store plane
QS(N)_CF	Q plane = (inverted) store plane, C plane = zeros
CQPQS(N)	Q plane = Q plane + (inverted) store plane, carry into C plane
CQPCQS(N)	Q plane = C plane + Q plane + (inverted) store plane, carry into C plane
CQPCS(N)	Q plane = C plane + (inverted) store plane, carry into C plane
CPQS(N)	C plane = carry from: Q plane + (inverted) store plane
CPCQS(N)	C plane = carry from: C plane + Q plane + (inverted) store plane
CPCS(N)	C plane = carry from: C plane = (inverted) store plane

6.2.3 Group 1

General characteristic: instructions fetch a store plane and equivalence or non-equivalence it with a specified bit of a specified MCU register (or of the edge register), storing the result in the A or Q plane.

AEBS(N)	A plane = (inverted) store plane EQUIV with register bit
AMEBS(N)	A plane = A plane ANDed with operand above
QEBS(N)	Q plane = (inverted) store plane EQUIV with register bit

6.2.4 Group 2

General characteristic: instructions load an MCU register (or the edge register) with a word, row or column from array store, or load an MCU register with an address.

RAW	Load address of word into MCU register
RAX	Load address of row into MCU register
RS	AND all rows of store plane into MCU or edge register
RSO	AND all columns of store plane into MCU or edge register
RW	Load MCU register with word from store plane
RWO	Load MCU register with orthogonal word from store plane
RX	Load MCU or edge register with store row
RXO	Load MCU or edge register with store column

6.2.5 Group 3

General characteristic: instructions broadcast the contents of an MCU register (or the edge register) to the columns of a PE register plane; that is, copy the orthogonal R plane into the destination plane.

AR(N)O	Set A plane equal to (inverted) orthogonal R plane
AMR(N)O	A plane = A plane ANDed with (inverted) orthogonal R plane
QR(N)O	Q plane = (inverted) orthogonal R plane
QPQR(N)O	Q plane = Q plane + (inverted) orthogonal R plane
QPCQR(N)O	Q plane = C plane + Q plane + (inverted) orthogonal R plane
QPCR(N)O	Q plane = C plane + (inverted) orthogonal R plane
CQPQR(N)O	Q plane = Q plane + (inverted) orthogonal R plane, carry to C plane
CQPCQR(N)O	Q plane = C plane + Q plane + (inverted) orthogonal R plane, carry to C plane
CQPCR(N)O	Q plane = C plane + (inverted) orthogonal R plane, carry to C plane
CPQR(N)O	C plane = carry from: Q plane + (inverted) orthogonal R plane
CPCQR(N)O	C plane = carry from: C plane + Q plane + (inverted) orthogonal R plane
CPCR(N)O	C plane = carry from: C plane + (inverted) orthogonal R plane

6.2.6 Group 4

General characteristic: instructions use an operand plane of all zeros (false) or all ones (true), and combine it with PE register planes or write it to store.

AF	A plane = zeros
AT	A plane = ones
QF	Q plane = zeros
QT	Q plane = ones
QQN	Invert Q plane
QPCQ	Q plane = C plane + Q plane
QPCQT	Q plane = C plane + Q plane + ones
QC	Q plane = C plane
QF_CF	Q plane, C plane = zeros
QT_CF	Q plane = ones, C plane = zeros
CQ_QQN	C plane = Q plane, invert Q plane
CQPCQ	Q plane = C plane + Q plane, carry to C plane
CQPCQT	Q plane = C plane + Q plane + ones, carry to C plane
QC_CF	Q plane = C plane, C plane = zeros
QCN	Q plane = inverted C plane
CF	C plane = zeros
CQ	C plane = Q plane
CPCQ	C plane = carry from: C plane + Q plane
CPCQT	C plane = carry from: C plane + Q plane + ones
SF	Store plane = zeros
XF	Store row = zeros
WF	Store word = zeros

QT_AT	Both A plane and Q plane = ones
QF_AF	Both A plane and Q plane = zeros

6.2.7 Group 5

General characteristic: instructions load into all bits of the A or Q plane a specified bit from a specified MCU register or the edge register.

AB(N)	All bits in A plane = (inverse of) selected bit
AMB(N)	All bits in A plane ANDed with (inverse of) selected bit
QB(N)	All bits in Q plane = (inverse of) selected bit

6.2.8 Group 6

General characteristic: instructions broadcast an MCU register or the edge register to the rows of a store plane; that is, copy the R plane into the destination plane.

SR(N)	Store plane = (inverted) R plane
XR(N)	Store row = (inverted) row of R plane
WR(N)	Store word = (inverted) register

6.2.9 Group 7

General characteristic: instructions broadcast an MCU register or the edge register to the rows of a PE register plane; that is, copy the R plane into the desination plane.

AR(N)	Set A plane equal to (inverted) R plane
AMR(N)	A plane = A plane ANDed with (inverted) R plane
QR(N)	Q plane = (inverted) R plane
QPQR(N)	Q plane = Q plane + (inverted) R plane
QPCQR(N)	Q plane = C plane + Q plane + (inverted) R plane
QPCR(N)	Q plane = C plane + (inverted) R plane
CQPQR(N)	Q plane = C plane + (inverted) R plane, carry to C plane
CQPCQR(N)	Q plane = C plane + Q plane + (inverted) R plane, carry to C plane
CQPCR(N)	Q plane = C plane + (inverted) R plane, carry to C plane
CPQR(N)	C plane = carry from: Q plane + (inverted) R plane
CPCQR(N)	C plane = carry from: C plane + Q plane + (inverted) R plane
CPCR(N)	C plane = carry from: C plane + (inverted) R plane

6.2.10 Group 8

General characteristic: instructions copy a suitably modified Q plane to a PE register plane, array store plane or row, or an MCU register or the edge register.

AQ(N)	A plane = (inverted) Q plane
AMQ(N)	A plane = A plane ANDed with (inverted) Q plane
RQO	MCU or edge register = logical AND of Q plane columns
SQ	Store plane = Q plane
SQ_AQ	Store plane and A plane = Q plane
SQ_CQ	Store plane and C plane = Q plane
SQ_QC(N)	Store plane = Q plane, Q plane = (inverted) C plane
SQ_QF	Store plane = Q plane, Q plane = zeros
SQ_QT	Store plane = Q plane, Q plane = ones
XQ	Store row = corresponding row from Q plane

6.2.11 Group 9

General characteristic: instructions copy a suitably modified A plane to another PE register plane, an array store plane, an MCU register or the edge register.

QA(N)	Q plane = (inverted) A plane
QPQA(N)	Q plane = Q plane + (inverted) A plane
QPCQA(N)	Q plane = C plane + Q plane + (inverted) A plane
QPCA(N)	Q plane = C plane + (inverted) A plane
QA(N)_CF	Q plane = (inverted) A plane, C plane = zeros
CQPQA(N)	Q plane = Q plane + (inverted) A plane, carry to C plane
CQPCQA(N)	Q plane = Q plane + C plane + (inverted) A plane, carry to C plane
CQPCA(N)	Q plane = C plane + (inverted) A plane, carry to C plane
CPQA(N)	C plane = carry from: Q plane + (inverted) A plane
CPCQA(N)	C plane = carry from: C plane + Q plane + (inverted) A plane
CPCA(N)	C plane = carry from: C plane + (inverted) A plane
RANO	MCU register or edge register = logical AND of inverted A plane columns
SAN	Store plane = inverted A plane
XAN	Store row = inverse of corresponding row from A plane

6.2.12 Group 10

General characteristic: instructions add to an array store plane under *activity control*; that is, only for those bits in the array store plane where the corresponding bits of the A plane are true (one). In some cases the Q or C plane is also written to, in which case updating of that plane takes place for all bits in the plane.

SIPQS	Store plane = Q plane + store plane
SIPCQS	Store plane = C plane + Q plane + store plane
SIPCS	Store plane = C plane + store plane
SIQPQS	Both Q plane and store plane = Q plane + store plane
SIQPCQS	Both Q plane and store plane = Q plane + C plane + store plane
SIQPCS	Both Q plane and store plane = C plane + store plane
SICQPQS	Both Q plane and store plane = Q plane + store plane, carry to C plane

SICQPCS	Both Q plane and store plane = C plane + store plane, carry to C plane
SICQPCQS	Both Q plane and store plane = C plane + Q plane + store plane, carry to C plane
SICPQS	Store plane = Q plane + store plane, carry to C plane
SICPCQS	Store plane = C plane + Q plane + store plane, carry to C plane
SICPCS	Store plane = C plane + store plane, carry to C plane

6.2.13 Group 11

General characteristic: instructions write to an array store plane under *activity control*; that is, only where corresponding bits of the A plane are true (one).

SIF	Store plane = zeros
SIQ	Store plane = Q plane
SIC	Store plane = C plane
SIPCQ	Store plane = Q plane + C plane
XIF	Store row = zeros
XIQ	Store row = Q plane
XIC	Store row = C plane
XIPCQ	Store row = Q plane + C plane

6.2.14 Group 12

General characteristic: instructions shift the PE register plane as specified by the effective *direction*, *geometry* and *count* values.

AQ	A plane = Q plane shifted one place
AQ_QQ	Both A plane and Q plane = Q plane shifted *count* places
QQ	Q plane = Q plane shifted by *count* places
AMQ	A plane = A plane ANDed with Q plane shifted one place
AMQ_QQ	Repeat AMQ and QQ (both shifted one place) a total of *count* times

6.2.15 Group 13

General characteristic: instructions add the rows or columns of the Q and C planes as if each were a set of edge-sized unsigned integers. The results of the addition are placed in the rows or columns of either or both of the Q and C planes. The action of the instruction depends on the effective *direction*, *geometry* and *count* values.

QVCQ	Q plane and C plane rows or columns added, sum in Q plane
CQVCQ	Q plane and C plane rows or columns added, sum in Q plane, carry in C plane
CVCQ	Q plane and C plane rows or columns added, carry in C plane

6.2.16 Group 14

General characteristic: instructions operate on MCU registers, and are mostly concerned with two MCU registers specified in the instruction and denoted below by R and M.

ADD	R = R + M
ADDC	R = R + M + C-flag
SUB	R = R – M
SUBC	R = R – M – inverted C-flag
ADDH	R = R + literal
ADDHC	R = R + literal + C-flag
SUBH	R = R – literal
SUBHC	R = R – inverted C-flag – literal
INCR	R = R + 1
DECR	R = R – 1
AND	R = (inverted) R AND (inverted) M
NAND	R = (inverted) R NAND (inverted) M
OR	R = (inverted) R OR (inverted) M
NOR	R = (inverted) R NOR (inverted) M
RR	R = M
RRN	R = inverse of M
RF	R = zeros
RT	R = ones
EQV	R = R EQUIV M
NEQ	R = R NOT.EQUIV M
ANDH	R = (inverted) R AND literal
ANDHN	R = (inverted) R AND (NOT literal)
NANDH	R = (inverted) R NAND literal
NANDHN	R = (inverted) R NAND (NOT literal)
ORH	R = (inverted) R OR literal
ORHN	R = (inverted) R OR (NOT literal)
NORH	R = (inverted) R NOR literal
NORHN	R = (inverted) R NOR (NOT literal)
RH	R = literal
RHN	R = inverted literal
EQVH	R = R EQUIV literal
NEQH	R = NOT (R EQUIV literal)
SHL	R = M shifted left *count* times
SHR	R = M shifted right *count* times
SHLC	R = M cyclic shift left *count* times
SHRA	R = M arithmetic shifted right *count* times
SHRC	R = M cyclic shift right *count* times
MPY32	R = least significant 32 bits from R x M (signed integers)
MPY32V	As above, but set V-flag for overflow
MPY64	Register pair (R – 1, R) = 64-bit product R x M (signed integers)
MPYU32	R = least significant 32 bits from R x M (unsigned integers)
MPYU32V	As above, but set V-flag for overflow
MPYU64	Register pair (R – 1, R) = 64-bit product R x M (unsigned integers)

6.2.17 Group 15

General characteristic: instructions provide branch and other control commands.

CALL	Supervisor entry
DO	Initiate instruction loop
EXIT	Return from subroutine
J	Jump
JE	Jump to another code section
JSL	Jump and preserve link in register - used for subroutine call
JESL	Subroutine call to another code section
NULL	No effect
SKIP	Skip next instruction on value of specified register bit
SKIP ALL	Skip next instruction on value of all bits of specified register
SKIP ANY	Skip next instruction on value of any bit of specified register
SKIP C	Skip next instruction on value of C-flag
SKIP V	Skip next instruction on value of V-flag

6.2.18 Pseudo instructions

LOOP	Signify end of DO LOOP (no code is generated)
PAUSE	Temporarily suspend program
RAC	Load address of instruction in code store
RACE	Load address of instruction in another code section
RALITR	Load address of first row of row-aligned literal
RALITW	Load address of first word of word-aligned literal
RAPL	Load address of plane-aligned data item
RAR	Load address of row of data
RASC	Load address of data section
RAWD	Load address of word of data
RDGC	Load direction, geometry and count modifier value
RLIT	Load literal value
STOP	Abandon program

6.2.18.1 Timing of instructions

Most instructions are executed in one cycle. The main exceptions are given in the table on the next page.

Type of instruction	*Cycles to execute*
Activity-controlled operations (groups 10 and 11)	2
Data returned from array memory (group 2)	3
Data returned from PE register (some cases of groups 8, 9)	2
Array shifts (group 12)	1 per 1-bit shift
Vector adds (group 13)	9 for DAP 500
	17 for DAP 600
MCU scalar multiply (group 14)	3 or 4
DO, jump (group 15)	2
EXIT (group 15)	3

Chapter 7

Addressing

This chapter is concerned with the generation of addresses (or related values) by the MCU while instructions are being executed: section 7.1 describes the different possible modes of calculation of these addresses or values; section 7.2 describes the format of addresses or values held in MCU registers that can contribute to address generation; and section 7.3 describes the syntax and usage of addressing constructs.

Of necessity, parts of this chapter will discuss the individual bits of an instruction word; the 'meaning' of a particular bit in an APAL instruction will depend on the range of machines the code will be assembled to run on. To enable you to see at a glance the bit structure for instructions for the different ranges, the relevant lines are tagged with 'DAP 500', 'DAP 600' or 'All ranges', as appropriate, or suitable comments are made in the flow of the text. Alternatively, some feature of an instruction, for example, may be described as only applicable to machines with *ES* greater than 32, that is machines with array edge sizes greater than DAP 500's.

7.1 Addressing modes

Six modes of addressing are catered for in the APAL instruction set. The different modes allow the MCU to construct one or more addresses or values by adding terms derived from:

- Fields in the instruction
- The DO loop iteration number
- A value held in a modifier register

Addressing modes A, B and C described below are generally used to address array store. Instructions concerned with accessing a complete array store plane will use mode A addressing, and will have an S in their instruction mnemonics; instructions concerned with accessing an array store row will use mode B addressing and have an **X** in their mnemonics; instructions concerned with accessing a word of an array store row will use mode C addressing, and have an **W** in their mnemonics.

Mode A addressing is also used where an instruction wants to address a bit in an MCU register, or the edge register; such instructions have a **B** in their mnemonics.

Modes A, B and C addressing

Modes A, B and C can be described as generating the *effective* ADDR, INT and WORD fields:

- The effective ADDR field specifies the address in the array store of the bit-plane of interest
- The effective INT field usually specifies the row or column address of interest in an ADDR-selected plane. Some instructions use the INT field to select one bit of the edge register, or one bit of an MCU register; in this latter case modulo 32 of the full INT field is used for bit selection
- The effective WORD field is only relevant with mode C addressing and in code assembled to run on a $ES > 32$ machine; the field specifies the address within a row or column of a 32-bit word

The effective ADDR, INT and WORD addresses are generated at run time from, amongst the other things mentioned above, ADDR, INT and WORD information supplied by you the user as part of the instruction. The method of calculating the effective ADDR, INT and WORD fields from the ADDR, INT and WORD fields in the instruction is described in sections 7.1.1, 7.1.2 and 7.1.3.

With certain instructions using mode A addressing, one or both of the ADDR and INT fields need not be specified. As a result, not all the bits in the instruction word (a fixed length of 32 bits) are necessarily defined; any undefined bits are disregarded. Appendix F discusses in more detail all the different instructions, including their usage of the various addressing modes.

The effective address calculated using modes A, B and C is a 29-bit value with a 'layout' of its bits of:

```
Z ZZZA AAAA AAAA AAAA AAAA AAAI IIII          DAP 500
Z ZAAA AAAA AAAA AAAA AAAA AIII IIIW          DAP 600
```

where

`Z...` are bits checked by the MCU. For instructions that access the array store, if any of these bits is not zero, a run-time error is flagged

`A...` is the effective ADDR field. For instructions that access the array store, the MCU flags a run-time error if the ADDR field is greater than the value held in the program limit register

`I...` is the effective INT field

`W` is the effective WORD field

Other addressing modes

Address mode D is concerned with parameters for array shift and vector add intructions, rather than array store addressing. Mode E is used to evaluate the count in a DO loop. Mode F is concerned only with calculation of the return address when control returns from a sub-routine as a result of executing an **EXIT** instruction.

7.1.1 Mode A : ADDR, INT evaluated separately

Mode A addressing is interpreted by the MCU in one of the following three ways, depending on which instruction is being processed:

- The effective ADDR field specifies a store plane; the effective INT field is discarded
- The effective INT field specifies a bit from either an MCU register or the edge register; the effective ADDR field is discarded
- The effective ADDR field specifies a store plane and the effective INT field specifies a bit from the MCU or edge registers

The effective ADDR and INT fields, where used, refer to separate entities. These fields are evaluated separately, although in the detailed description below these fields are shown concatenated in order to emphasize the similarity with modes B and C addressing, described in sections 7.1.2 and 7.1.3. In $ES > 32$ machines, a WORD field is catered for in some instructions; mode A makes no use of the field.

The effective address fields are constructed using 29-bit arithmetic, with any carry-out from the most significant bit position being ignored. In this address mode, there is also no carry propagation between the INT and ADDR parts of the field.

Up to four of the following four terms may be added together to produce the effective ADDR and INT fields; the actual fields that are added depending on the instruction concerned:

1 A composite instruction address field:

```
0 0000 0000 0000 000A AAAA AAAI IIII          DAP 500
0 0000 0000 0000 0AAA AAAA AIII III0          DAP 600
```

where

`A`... is the ADDR field you supplied in the instruction
`I`... is the INT field you supplied in the instruction

2 A field derived from the current DO loop iteration number (for INT field stepping):

```
0 0000 0000 0000 0000 0000 000N NNNN          DAP 500
0 0000 0000 0000 0000 0000 0NNN NNN0          DAP 600
```

where `N`... are the least significant bits of the current DO loop iteration number

This term is only added when stepping is applicable, that is, if the instruction is inside a DO loop, and the instruction specifies that the address is to be stepped. A bit in the INCREMENT/DECREMENT field specifies if the DO loop steps are to be negative, in which case this second term will be subtracted from term 1

3 Another field derived from the current DO loop iteration number (for ADDR field stepping):

```
N NNNN NNNN NNNN NNNN NNNN NNN0 0000        DAP 500
N NNNN NNNN NNNN NNNN NNNN N000 0000        DAP 600
```

where N... are the least significant bits of the current DO loop iteration number

This term is only added when stepping is applicable, that is, if the instruction is inside a DO loop, and the instruction specifies that the count value is to be stepped. A bit in the INCREMENT/DECREMENT field specifies if the DO loop steps are to be negative, in which case this second term will be subtracted from term 1

4 Modifier:

```
M MMMM MMMM MMMM MMMM MMMM MMMM MMMM        All ranges
```

where M... are the least significant 29 bits of the contents of the modifier register (see section 7.2.1)

If no modification is specified in the instruction, this fourth term is not added

Note that for any particular instruction, terms 2 and 3 (if relevant) can only be both added, both subtracted, or both ignored.

Mode A addressing caters for addressing a store plane, or addressing a bit in the edge register or an MCU register, or both plane and bit; the **QEBS** instruction, for example, addresses both.

The following addition illustrates how on a DAP 500 all of the above four possible terms could be added to form the composite address fields for a **QEBS** instruction:

```
0 0000 0000 0000 000A AAAA AAAI IIII
0 0000 0000 0000 0000 0000 000N NNNN
N NNNN NNNN NNNN NNNN NNNN NNN0 0000
M MMMM MMMM MMMM MMMM MMMM MMMM MMMM
------------------------------------
V VVVV VVVV VVVV VVVV VVVV VVVV VVVV
```

which would be interpreted as:

```
Z ZZZA AAAA AAAA AAAA AAAA AAAI IIII
```

The addition for the same instruction running on **DAP 600** would be:

```
0 0000 0000 0000 0AAA AAAA AIII III0
0 0000 0000 0000 0000 0000 0NNN NNN0
N NNNN NNNN NNNN NNNN NNNN N000 0000
M MMMM MMMM MMMM MMMM MMMM MMMM MMMM
------------------------------------
V VVVV VVVV VVVV VVVV VVVV VVVV VVVV
```

which would be interpreted as:

```
Z ZAAA AAAA AAAA AAAA AAAA AIII IIIW
```

where:

- `V`... is the value of the addition of the terms
- `Z`... are bits checked by the MCU
- `A`... is the effective ADDR field
- `I`... is the effective INT field
- `W` is the effective WORD field (which is ignored for mode A addressing)

Note, as stated above, any carry-out from the left-hand bit (that is, the most significant of the 29 bits) is ignored, as is any carry across the boundary between the the ADDR and INT fields.

The 29-bit result was interpreted as stated in section 7.1. Note that if the array is being addressed, and the first few most significant bits – the ones labelled `z`... above – are not zero, then a run-time error will occur.

To illustrate how plane and bit address stepping works in mode A, suppose that on the first pass of a DAP 500 DO loop, instruction **QEBS** generates an effective address of 60.29 (that is, bit plane 60, register bit number 29). If address incrementing is specified, then on subsequent passes of the loop the effective address will be 61.30, 62.31, 63.0, 64.1, and so on. In a similar DAP 600 DO loop, if the starting address was 65.62, then the addresses in subsequent passes of the loop would be 66.63, 67.0, 68.1, and so on.

Other instructions using mode A addressing include **CQPQS**, **QS** and **AMB**.

7.1.2 Mode B : carry propagates from INT to ADDR

Instructions that access an array store row or column use mode B addressing, in which the effective ADDR specifies the store plane, and the effective INT specifies the row or column within that plane. These instructions have an **X** in their mnemonics. Mode B addressing allows modification and DO loop stepping across plane boundaries. For example on DAP 500, row 31 of plane 10 and row 0 of plane 11 are regarded as consecutive rows. Hence in mode B addressing, the ADDR and INT fields are concatenated, with carry-outs being propagated from INT to ADDR. As with mode A addressing, for code running on $ES > 32$ machines, the WORD field is not used.

The address fields are constructed using 29-bit arithmetic, with any carry-out from the most significant bit position being ignored. Unlike mode A addressing, during the addition that constructs the effective ADDR and INT fields any carry bit from the most significant bit in the INT field is carried forward to the least significant bit of the ADDR field.

Up to three of the following three terms are added together to form the effective ADDR and INT fields:

1 A composite instruction address field:

```
0 0000 0000 0000 000A AAAA AAAI IIII                DAP 500
0 0000 0000 0000 0AAA AAAA AIII IIIO                DAP 600
```

where

- `A`... is the ADDR field you supplied in the instruction
- `I`... is the INT field you supplied in the instruction

2 Either of:

a A field derived from the current DO loop iteration number (for stepping the concatenated INT and ADDR fields):

```
N NNNN NNNN NNNN NNNN NNNN NNNN NNNN          DAP 500
N NNNN NNNN NNNN NNNN NNNN NNNN NNNO          DAP 600
```

where N... are the least significant 29 or 28 bits of the DO loop iteration number

b A field derived from the current DO loop iteration number (for stepping the ADDR field only):

```
N NNNN NNNN NNNN NNNN NNNN NNNO 0000          DAP 500
N NNNN NNNN NNNN NNNN NNNN NOOO 0000          DAP 600
```

where N... are the least significant 24 or 22 bits of the DO loop iteration number.

The STEP TYPE instruction field specifies which of the above alternative stepping terms is used.

The addition of the appropriate stepping term only occurs when stepping is applicable. That is, if the instruction is inside a DO loop, and the instruction specifies that the address is to be stepped. A bit in the INCREMENT/DECREMENT field specifies if the DO loop steps are to be negative, in which case the selected stepping term will be subtracted from term 1

3 Modifier

```
M MMMM MMMM MMMM MMMM MMMM MMMM MMMM          All ranges
```

where M... are the least significant 29 bits of the contents of the modifier register (see section 7.2.1).

If no modification is specified in the instruction, this third term is not added

In a similar way to mode A addressing, a 29 bit sum is formed from whichever of the above three terms is relevant; any carry out from the 29th least significant bit position is ignored. As with mode A, the 29 bit value is interpreted and checked as described in section 7.1.

For example, suppose in a DAP 600 machine on the first pass of a DO loop an instruction using mode B addressing (instruction **XAN** perhaps) is operating on an effective address of 50.2 (that is, bit plane 50, row 2). If decrementing the row address is specified, then on subsequent passes of the DO loop the effective address would be 50.1, 50.0, 49.63, 49.62, and so on. If, instead of decrementing the row address, incrementing the bit-plane address was specified, then on subsequent passes of the DO loop the effective address would be 51.2, 52.2, 53.2, and so on.

Other instructions using mode B addressing include **RX**, **XIQ**.

7.1.3 Mode C : word address

Instructions that access words in an array store plane have a **W** in their mnemonics and use mode C addressing, which is similar to mode B addressing; in mode C use is made of the WORD field to specify the word of interest.

The address fields are constructed using 29-bit arithmetic, with any carry-out from the most significant bit position being ignored. In a similar way to the process used in mode B addressing, during the addition that constructs the effective ADDR, INT and WORD fields any carry bit from the most significant bit in the WORD field is carried forward to the least significant bit of the INT field, and any carry bit from the most significant bit in the INT field is carried forward to the least significant bit of the ADDR field.

On DAP 500 mode C is identical to mode B. However, if you want your code to be portable between all ranges of DAP, even when you are programming for a DAP 500 you should distinguish between word accesses and row accesses. For completeness, mode C instructions to run on both DAP 500 and DAP 600 are considered here.

Up to three of the following three terms are added together to form the effective ADDR, INT and WORD fields:

1 A composite instruction address field:

```
0 0000 0000 0000 000A AAAA AAAI IIII          DAP 500
0 0000 0000 0000 0AAA AAAA AIII IIIW          DAP 600
```

where

`A`... is the ADDR field you supplied in the instruction
`I`... is the INT field you supplied in the instruction
`W` is the WORD field you supplied in the instruction

2 A field derived from the current DO loop iteration number (for stepping the concatenated WORD, INT and ADDR fields):

```
N NNNN NNNN NNNN NNNN NNNN NNNN NNNN          All ranges
```

where `N`... are the least significant 29 bits of the DO loop iteration number

The addition of the appropriate stepping term only occurs when stepping is applicable. That is, if the instruction is inside a DO loop, and the instruction specifies that the address is to be stepped. A bit in the INCREMENT/DECREMENT field specifies if the DO loop steps are to be negative, in which case the selected stepping term will be subtracted from term 1.

Note that in mode C addressing, DO-loop stepping of the ADDR field only is not allowed, unlike in mode B addressing

3 Modifier

```
M MMMM MMMM MMMM MMMM MMMM MMMM MMMM          All ranges
```

where `M`... are the least significant 29 bits of the contents of the modifier register (see section 7.2.1).

If no modification is specified in the instruction, this third term is not added

As with modes A and B addressing, a 29 bit sum is formed from whichever of the above terms is relevant; any carry out from the 29th least significant bit position is ignored; the 29 bit value is interpreted and checked as described in section 7.1.

For example, suppose in a DAP 600 machine on the first pass of a DO loop an instruction using mode C addressing is concerned with an effective address of 38.62.1 (that is, bit plane 38, row 62, word 1). If incrementing the address is specified, then on subsequent passes of the DO loop the effective address would be 38.63.0, 38.63.1, 39.0.0, and so on.

Instructions using mode C addressing include **RW** and **WR**.

7.1.4 Mode D : direction, geometry and count

Mode D is concerned with specifying four parameters: the direction of shift in an array shift operation; the direction of carry propagation for an array vector add operation; the 'geometry' or the edge effect for such operations; and the count, or number of planes of shift or carry propagation. Many instructions that use mode D addressing have a Q in their mnemonics, as they are, in part at least, concerned with the Q plane.

The effective direction of shift or carry propagation is formed using the DIRECTION field in the instruction, and optionally a field from a modifier register. The options available are:

- North
- East
- South
- West

$\Big\}$ independent of the contents of the modifier register, even if a modifier register is specified

- Direction specified by part of the contents of the modifier register
- Direction specified by part of the contents of the modifier register, with the addition of 90 degrees clockwise rotation
- Direction specified by part of the contents of the modifier register, with the addition of 180 degrees clockwise rotation
- Direction specified by part of the contents of the modifier register, with the addition of 270 degrees clockwise rotation

The effective geometry specifies the edge effect, as discussed in section 6.1.6, and is formed using the instruction GEOMETRY field, or a field from a modifier register.

The options available are :

- Plane EW, Plane NS
- Plane EW, Cyclic NS
- Cyclic EW, Plane NS
- Cyclic EW, Cyclic NS
- Geometry specified by the contents of the modifier register

Note that where the EW and NS geometries differ, then the effective geometry depends on the effective direction, as defined above.

The effective count value is 5 bits wide for DAP 500, and 6 bits wide for DAP 600. The value is constructed by taking the sum of up to 3 of the following terms:

1 The value you supplied in the INT field of the instruction

2 A field derived from the current DO loop iteration number (for stepping the COUNT value).

The addition of the appropriate stepping term only occurs when stepping is applicable. That is, if the instruction is inside a DO loop, and the instruction specifies that the count value is to be stepped. A bit in the INCREMENT/DECREMENT field specifies if the DO loop steps are to be negative, in which case the selected stepping term will be subtracted from term 1

3 The least significant bits of the contents of a modifier register.

If no modification is specified in the instruction, this third term is not added

The effective direction and geometry are unaffected by DO loop stepping.

Instructions using mode D addressing include **QQ**, **CVCQ**, and **QVCQ**.

7.1.5 Mode E : DO loop count

The value of the DO loop iteration count is given by adding the instruction DO COUNT field, and (if instruction modification is specified) the entire contents of the modifier register.

If the loop count is unmodified, then the count values that can be encoded in the instruction are in the range 1 to 256.

If the loop count is modified, then the count value encoded in the instruction is a signed offset in the range −255 to +256. If the result of such count modification is zero then the effect is undefined.

7.1.6 Mode F : EXIT instruction

In the **EXIT** instruction, the address in code store of the destination instruction is evaluated by adding the instruction CODE ADDRESS field to the least significant 20 bits of the modifier register, ignoring any carry-out.

7.2 Modifier register formats

This section describes the various formats for the contents of MCU registers when they are used as modifier registers.

Array instructions only permit M1 to M7 to be used as modifiers, but for the DO count and for instructions **J** and **JE**, M1 to M13 are usable. Any MCU register (including M0) can be used with the **EXIT** instruction to construct the destination instruction address.

7.2.1 Array store plane and row, column, word or bit number modifier

The modifier always has the format of a word address, as follows:

```
XXOO OOOA AAAA AAAA AAAA AAAA AAAI IIII        DAP 500
XXOO OAAA AAAA AAAA AAAA AAAA AIII IIIW        DAP 600
```

where

`X`... are bits whose values are disregarded by the MCU (often called 'don't care' bits)
`A`... is the ADDR field, corresponding to a plane address
`I`... is the INT field, corresponding to row or column number or register bit number
`W` is the WORD field, corresponding to a word address within a row

Such modifiers can be constructed using the instructions **RAX** or **RAW**, or pseudo-instructions **RASC**, **RAPL**, **RAR** or **RAWD**.

When you use such a modifier to generate an array store address, the effective address is subject to the checks described in section 7.1. Hence some of the most significant bits in the modifier should be zero (bits 3 to 6 inclusive for DAP 500, bits 3 and 4 for DAP 600). Also, when you address the array store, bit 2 in the modifier should be zero, otherwise a run-time error will result.

7.2.2 Direction, geometry and count modifier

Modifiers of this form can be used by the array shift and vector add instructions.

The format of the modifier register is:

```
XXXX XXXX XXXX XXXX XXDD XXGG XXXC CCCC        DAP 500
XXXX XXXX XXXX XXXX XXDD XXGG XXCC CCCC        DAP 600
```

where

`X`... are bits whose values are disregarded by the MCU (often called 'don't care' bits)
`DD` is the direction field
`GG` is the geometry field
`C`... is the count field

Possible values for DD and GG are :

DD	00	North	GG	00	Plane	EW,	Plane	NS
	01	East		01	Plane	EW,	Cyclic	NS
	10	South		10	Cyclic	EW,	Plane	NS
	11	West		11	Cyclic	EW,	Cyclic	NS

Section 7.1.4 specifies direction and geometry modification; even when a modifier is specified, the direction and/or geometry fields of the modifier register can be ignored.

You are recommended to construct these modifiers using the pseudo-instruction **RDGC**.

7.2.3 DO count modifier

A modifier of this type can be used in constructing the effective count for a **DO** loop (see section 7.1.5):

```
TTTT TTTT TTTT TTTT TTTT TTTT TTTT TTTT                    All ranges
```

where `T...` is the field in the modifier which will be added to the 'times round the DO-loop' count.

Note that the entire contents of the modifier register are used.

7.2.4 Instruction address modifier

Modifiers of this type are used with the **EXIT** or **J** instructions.

The format of the register is:

```
XXXX XXXX XXXX RRRR RRRR RRRR RRRR RRRR                    All ranges
```

where

`X...` are bits whose values are disregarded by the MCU (often called 'don't care' bits)
`R...` is the modifier field to be added to the return instruction address

These modifiers are created typically when a **JSL** or **JESL** instruction is used to enter a subroutine, or when a **RAC** or **RACE** instruction is used. Note that in either case the most significant bits are undefined, and are ignored by **EXIT** or **J**.

7.3 Addressing constructs

This section describes the syntax and usage of a number of addressing constructs that are used in APAL instructions.

7.3.1 Specifying modifier registers

Modifiers must be used to form the following array store addresses:

- An address of a plane, row or word in a data section
- An address of a plane, row or word in the stack section

Use of a modifier to form or select any of the following items is optional:

- An address of a plane, row or word in the workspace area (planes 0 to 119)
- A bit number in an MCU register or the edge register
- DIRECTION, GEOMETRY and COUNT fields
- DO loop iteration count
- Instruction address

For the DO loop iteration count, or instruction addresses the modifier is specified as follows:

<doj modifier> ::= (<dojmreg>)

<dojmreg> ::= M1 | M2 | M3 | M4 | M5 | M6 | M7 | M8 | M9 | M10 | M11 | M12 | M13

All other types of modifier are as follows:

<modifier> ::= (<mreg>)

<mreg> ::= M1 | M2 | M3 | M4 | M5 | M6 | M7

If DO loop stepping is also specified, the modifier and step can be written in a short combined form (see section 7.3.3).

7.3.2 Specifying DO loop stepping

For most instruction types, stepping of addresses or values is specifed as:

<step> ::= (+) | (–)

These step options are the only ones available with mode A or mode C addressing (see sections 7.1.1 and 7.1.3 for more details).

For those instructions that access a row or column of array store, the stepping is specified as:

<step A> ::= (+A) | (–A) | <step>

These step options are used with mode B addressing (see section 7.1.2 for more details). Use of options +A or –A cause the bit-plane address to be stepped; options + or – cause the row or column address to be stepped.

7.3.3 Array store addresses

The syntax of array store addresses is:

<array store address> ::= <store plane address> | <store row address> |
<store column address> | <store word address>

<store plane address> ::= <plane><modifier>?<step>?

<store row address> ::= <row><modifier>?<step A>?

<store column address> ::= <column><modifier>?<step A>?

<store word address> ::= <word><modifier>?<step>?

<data address> ::= <plane> | <row> | <column> | <word>

<plane> ::= <aligned data name><plane offset>? | <plane number>

<row> ::= <name or plane><row offset>? | <row offset>

<column> ::= <name or plane><column offset>? | <column offset>

<name or plane> ::= <data name><plane offset>? | <plane number>

<word> ::= <row> |
    <name or plane>?.<word offset> |
    <name or plane>?<row offset><word offset>

<aligned data name> ::= <data name>

<data name> ::= <data section name> | <data variable name> | <identity name>

<plane offset> ::= + <plane number>

<row offset> ::= .<numval>

<column offset> ::= .<numval>

<word offset> ::= .<numval>

<plane number> ::= <numval>

When <modifier> and <step> are both present in a store address, they can be combined by combining the pairs of brackets; for example, the following are equivalent:

```
(M1)(+)
(M1+)
```

Some examples of array addresses are:

```
DATAVAR1 + 14.2(M3)
DATAVAR2 + 10 (M4)(+)
20.14(-)
.9(M6)
1..3(M4)
```

Effective array store addresses are constructed at run time using mode A, mode B or mode C addressing (for more details see sections 7.1.1, 7.1.2 and 7.1.3 respectively), depending on the instruction concerned.

Note that when a data variable name, or a data identity name which represents an address within a data section is used, then a modifier register is mandatory; the modifier must contain the address of the start of the data section. Since the instruction ADDR field is eight bits, the value of <plane number>, or the instruction field implied by <aligned data name> together with <plane offset> must be in the range 0 to 255. The following program extract illustrates these points and the use of pseudo-instructions **RASC** and **RAPL** in constructing addresses:

```
DATA D
D1:256*PLANE
D2:1,2,3
.
.
```

```
.
END
CODE C
.
.
.
RASC    M1      D1                  ! M1 CAN BE USED TO ACCESS THE
                                    ! FIRST 256 PLANES OF D.
QS              D1(M1)              ! LOAD FIRST PLANE OF D.
QS              D1+255 (M1)         ! LOAD PLANE 255 OF D.
QS              D2(M1)              ! THIS WOULD FAIL AT ASSEMBLY TIME;
                                    ! AN OFFSET OF 256 WILL NOT FIT
                                    ! IN INSTRUCTION.
RAPL    M2      D2                  ! LOAD ADDRESS OF PLANE 256 OF D INTO M2.
QS              0(M2)               ! LOAD PLANE 256 OF D.
QS              255(M2)             ! LOAD PLANE 511 OF D.
.
.
END
```

7.3.4 Register bit addresses

Bit selection can be applied to an MCU register or to the edge register. The syntax is:

<MCU-or-edge-register bit address> ::= <MCU-or-edge-register>.<bit number><modifier>?<step>?

<MCU-or-edge-register> ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7 | M8 | M9 | M10 | M11 | M12 | M13 | ME

<bit number> ::= <numval>

The bit number must be in the range 0 to 31 for an MCU register, or 0 to (ES -1) for the edge register.

Some examples of register bit selection are:

M0.12
M11.20(M2)
M6.14(M3)(+)
ME.30(–)

Some APAL instructions address both a register bit and a store plane. The syntax of such an address is:

<MCU-or-edge-register>.<bit number><store plane address>

Some examples are:

M0.12 VARI + 2 (M6)(+)
M6.14 171(M4)

The effective bit number is the INT value constructed at run time using mode A addressing (see section 7.1.1. for more details). When a bit of an MCU register is being selected, modulo 32 of the effective INT value is used.

Note that the bit selection number in a modifier register occupies the INT field (as defined in section 7.1), which for machines with $ES > 32$ is not held in the least significant bits of the modifier. Hence you will often find it convenient to construct such modifiers using the **RAX** instruction – see appendix F for details.

7.3.5 Code store addresses

Much as with data store addresses, once a code store address has been given a label, then you can refer to that address simply by the label name.

Code store addresses within the same code section have the following syntax:

```
<within-section address> ::= <code label name><label offset>?<doj modifier>? |
                             <star><label offset><doj modifier>?
<label offset> ::= + <numval> | – <numval>

<star> ::= *
```

Thus a code store address can be either of the following:

- The name of a code label in the same code section, with an optional displacement (in instructions) forwards or backwards, and with an optional modifier
- A displacement, forwards or backwards, from the current instruction, with an optional modifier. The current instruction is represented by the character *

Any within-section code address can be referenced by a **J** instruction. Any unmodified address can be referenced by a **JSL** instruction or an **RAC** instruction. Typical instructions might be:

```
J              LAB1      ! jump to instruction labelled LAB1

JSL            LAB2 +3   ! jump to the third instruction after the one
                         ! labelled LAB2, saving a link value in M0

J              *–2       ! jump to the instruction two before the present one

RAC      M1    LAB2      ! load the address of LAB2 into M1
```

Code store addresses within a different code section have the following syntax:

```
<inter-section address> ::= <code section name><section offset>?<doj modifier>? |
                            <entry point name><section offset>?<doj modifier>?
<section offset> ::= + <numval>
```

Thus the address can be the name of another code section, or the name of an entry point within another code section, either of them having an optional forward displacement and an optional modifier.

Any of these address forms can be used with the **JE** instruction. The unmodified forms can be used with the **JESL** or **RACE** instructions.

When you are transferring control between sections, you should take account of the software conventions given in section 9.6.

The syntax of the EXIT instruction is:

```
EXIT <MCU register>?<offset>?
<offset> ::= <numval>
```

You can regard the MCU register (including M0, the default) as being a modifier for the offset.

Chapter 8

Tracing facilities

APAL provides a facility which you can use to suspend the execution of a DAP program temporarily, and output the contents of various registers and a segment of the array store part of your DAP program block. This facility is intended as an aid to program development; it will affect program performance adversely.

8.1 The TRACE statement

The tracing facility is provided by the TRACE statement.

8.1.1 Syntax

```
<trace_statement> ::= TRACE<trace_number>?<registers_trace_item><trace_level><newline> |
                      TRACE<trace_number>?<registers_trace_item>?<trace_level>
                                         <array_store_trace_item><newline>

<array_store_trace_item> ::= <word><modifier>?<trace_count>? WORDPACK?<type/size>?|
                             <word><modifier>?<trace_count>? ROWPACK<type/size>?
                                     <start_bit>?|
                             <word><modifier>?<trace_count>? VERTICAL<type/size>?
                                     <row_range>?<col_range>?|
                             <word><modifier>?<trace_count>? VERTICAL<type/size>?
                                     <col_range>?<row_range>?

<trace_number> ::= <numval>

<registers_trace_item> ::= PER|MER|PER MER|MER PER

<modifier> ::= (<mreg>)

<mreg> ::= M1 | M2 | M3 | M4 | M5 | M6 | M7

<trace_level> ::= LEVEL <numval>
```

<trace_count> ::= <numval>

<type/size> ::= <type><size>?

<type> ::= HEX|INT|REAL|CHAR|BIT

<size> ::= (<numval>)

<start_bit> ::= FROM_BIT <numval>

<row_range> ::= ROWS (<numval>,<numval>)

<col_range> ::= COLS (<numval>,<numval>)

8.1.2 Semantics

The APAL tracing facility is controlled by TRACE statements within an APAL code section. TRACE statements must not appear outside a code section. When the output is complete, execution continues with the next APAL instruction.

TRACE has the following basic form:

> TRACE *trace-number* *registers-trace-item* LEVEL *trace-level-number*
> *array-store-trace-item*

where:

> *trace-number* is an unsigned integer or hexadecimal value, or an assembly-time expression within [ ] yielding a non-negative integer value; it must lie in the range 0 - 1023, and appears at the head of the output generated by TRACE. *trace-number* is used to identify the origin of the TRACE output; you can omit *trace-number*, in which case it defaults to the line number of the TRACE statement generated for the APAL code listing
>
> *registers-trace-item* is any of MER, PER, MER PER, or PER MER. The effect is to TRACE the contents of the MCU and edge registers (MER), the PE registers – the A, Q and C planes of the PEs – (PER), or both the MCU and edge registers and the PE registers (MER PER, and PER MER). MER PER and PER MER have the same effect; they both specify an output of PER then MER. Either or both of *registers-trace-item* and *array-store-trace-item* (see below) can be included in a TRACE statement
>
> *trace-level-number* is an unsigned integer, a hexadecimal value, or an assembly-time expression in [ ] producing an integer in the range 1 to 15.
>
> At APAL source assembly time, you can specify a TRACE option with a value in the range 0 to 15. The effect of this assembly-time TRACE option is that any TRACE statement with a *trace-level-number* greater than your option will not be assembled. The default for this assembly-time option is no TRACEing.

You can specify a further TRACE option value, which has effect when the program is run. Only those assembled TRACE statements with a *trace-level-number* not exceeding the run-time option value will generate output. The default for this run time option is that output is produced for all assembled TRACE statements.

A detailed description of how to control TRACE at assembly and run time is given in *DAP Series: Program Development*

array-store-trace-item specifies the array store item(s) to be TRACEd, and is defined below. Either or both of *array-store-trace-item* and *registers-trace-item* can be included in a TRACE statement

8.1.2.1 Array store trace items

Unless you tell it otherwise, TRACE has no knowledge of the structure of the data item(s) to be traced. You must tell it therefore what to assume about the format of the items (integer, real, and so on), about the size of each item, about how many items are to be traced, about how they are packed in store, and so on. Your specification for *array-store-trace-item* gives TRACE this information.

array-store-trace-item can take any of the forms:

word (modifier) trace-count WORDPACK *type (size)*

word (modifier) trace-count ROWPACK *type (size) start-bit*

word (modifier) trace-count VERTICAL *type (size) row-range col-range*

where:

word (modifier) specifies an effective address, the word address of the start of the (first) data item to be TRACEd. This address is similar to an array store address (see section 7.3.3 for more details), except that:

- Address stepping is not allowed
- A global name can be specified without a modifier, since the address fields associated with TRACE are wide enough to address anywhere within the DAP program block (unlike the address fields of APAL instructions). If both *word* and (*modifier*) are specified, the two are added together to form the effective word address, using carry across the INT and WORD and ADDR and INT boundaries – as in Mode C addressing

trace-count is an unsigned integer, a hexadecimal value, or an assembly-time expression in [ ] producing an integer; it specifies how may items are to be traced out, and should lie in the range 1 to 2^{29}. The parameter is optional; if you do not specify one a default of 1 is assumed

WORDPACK, ROWPACK and VERTICAL specify the way the data is assumed to be mapped in the DAP memory, and are discussed below

type specifies the type to be assumed for the data item(s) to be TRACEd; it can take any one of the values given in the table on the next page *(size)* is an unsigned integer, hexadecimal value or an assembly-time expression in [ ].

The values of *type* and the ranges within which *size* should lie are given below:

Value of type	*Effect: possible range of values for* size
INT	Data is interpreted and output as two's complement integer values: 1 to 64 bits, in steps of 1 bit
REAL	Data is interpreted and output as real values: 24 to 64 bits, in steps of 8 bits
HEX	Data is interpreted and output as hexadecimal values: 1 bit to (either 64 for WORDPACK or VERTICAL formats; or for ROWPACK, 64 or *ES*, whichever is the greater), in steps of 1 bit
CHAR	Data is interpreted and output as ASCII character values (see appendix B for details): 24 bits to (either 64 for WORDPACK or VERTICAL formats; or for ROWPACK, 64 or *ES*, whichever is the greater) bits, in steps of 8 bits
BIT	Data is interpreted and output as bit patterns: 1 to (either 64 for WORDPACK or VERTICAL formats; or for ROWPACK, 64 or *ES*, whichever is the greater) bits, in steps of 1 bit

You can omit *type*, in which case a default value of HEX is assumed

If you specify *type*, specifying *(size)* is optional; if you do not specify *type*, specifying *(size)* will cause an assembly-time error. If you do not specify *(size)* it will default to 32 for all values of *type*

One word of caution: notice that TRACE's maximum size for a CHAR item is 64 bits, not the 512 bits of a normal data item. Should you want to TRACE character strings larger than 64 bits, then one way would be to TRACE several *ES*-sized WORDPACKed CHAR data items starting at the same address in store as each of your large strings

start-bit specifies the start of the data item(s) to be TRACEd as an offset from the start (the most significant bit) of the selected row(s), and has the form:

FROM_BIT *bit-number*

where *bit-number* can take any value from 0 to $ES - 1$. The sum of *bit-number* and *size* must not exceed *ES*. *start-bit* is optional; the default is such that each item is assumed to be 'right aligned' in its row; for example, in code to run on a DAP 600, if *type (size)* is INT (32), then the default for *start-bit* is 32

row-range specifies the range of rows to be output in each plane selected for TRACEing; it takes the form:

ROWS (*start-row*, *end-row*)

where *start-row* and *end-row* define the start and end of the blocks of rows of interest; each is in the range 0 to $ES - 1$. The parameter is optional; if you do not specify any ROWS a default of all rows is assumed. ROWS (...) and COLS (...) can be specified in either order

column-range specifies the range of columns to be output in each plane selected for TRACEing; it takes the form:

COLS (*start-column*, *end-column*)

where *start-column* and *end-column* define the start and end of the blocks of columns of interest; each is in the range 0 to $ES - 1$. The parameter is optional; if if you do not specify any COLS a default of all columns is assumed. COLS (...) and ROWS (...) can be specified in either order

8.2 Format of items to be TRACEd

For TRACE to produce output that can easily be understood by you, it needs to know how the data you want to TRACE is held in the array store. As explained in chapter 4, when you initialise a data section, you can state that data is to be packed into a series of contiguous horizontal words (WORDPACK), or into a series of contiguous horizontal rows (ROWPACK). Similarly, TRACE lets you specify that the data you want to TRACE is stored WORDPACKed or ROWPACKed. Although you cannot declare or initialise data in VERTICAL format (PLANE_ALIGN lets you declare and initialise data starting at a plane boundary, but the data is still stored horizontally), you can TRACE VERTICALly stored data.

The following sections discuss the three possible formats.

8.2.1 WORDPACK format

If you specify WORDPACK, TRACE assumes that each data item to be TRACEd is packed into a whole number of 32-bit words in array store; the *type (size)* specified in the statement and discussed above, specify the assumed details of the item(s) in the word(s).

8.2.2 ROWPACK format

If you specify ROWPACK, TRACE assumes that each data item to be TRACEd is packed into a whole number of rows in array store. If the effective word address you specified in the TRACE statement is not at a row boundary, then TRACEing starts at the beginning of the row that contains the effective word address; the specified or default value of *start-bit* may modify TRACE's starting point.

If you do not specify any *start-bit*, then TRACE will output the *type* and *(size)* of data item as specified (explicitly or by default), using a default value for *start-bit*, such that each data item is assumed to be 'right aligned' in its row.

You can use *start-bit* to TRACE unusual data structures in store, or part only of normal data items; one use might be to TRACE only the sign bits of an array of vectors.

8.2.3 VERTICAL format

If you specify VERTICAL, TRACE assumes that each data item to be traced is a vertical mode matrix (of size *ES* by *ES*, unless you specify either or both of 'COLS (*start-column*, *end-column*)' and 'ROWS (*start-row*, *end-row*)'). Each successive data item (if any) is a similar matrix, starting in array store, *size* bit planes from the previous item TRACEd.

If the effective word address you specified in the TRACE statement is not at a plane boundary, then TRACEing starts at the beginning of the plane that contains the effective word address, modified by any 'COLS (*start-column*, *end-column*)' and 'ROWS (*start-row*, *end-row*)' you specify.

8.3 Examples

- A simple example:

  ```
  TRACE  PER MER  LEVEL 5
  ```

  which will output, if traces of level 5 or lower are required, a trace report numbered with the line number of the TRACE statement in the listing of the APAL source code. The report will consist of the contents of the PE register planes, then the contents of the MCU registers.

- An example of tracing WORDPACKed data items:

  ```
  FRED = 24.3.0
  .
  .
  .
  TRACE 23 LEVEL 8 FRED 32 WORDPACK
  ```

  The TRACE statement will output a report numbered 23 if reports of level 8 or lower are required. The report will include 32 hexadecimal data items, each 32 bits wide, taken from array store starting at address 24.3.0

See the entry for TRACE in appendix F for further examples.

Chapter 9

Code section conventions

9.1 Introduction

This chapter describes:

- Entry and exit conventions for an APAL code section
- The areas of the DAP available to the code section
- How to call an APAL or a FORTRAN-PLUS subprogram from an APAL code section
- How to call an APAL code section from a host FORTRAN (or whatever) subprogram

9.1.1 Run-time structure

Any program that uses the DAP consists of a *DAP program* and an associated *host program*.

The host program consists of one or more modules commonly, but not necessarily, written in FORTRAN or C that execute on the host. The DAP program consists of one or more modules written in either or both of FORTRAN-PLUS and APAL that execute on the DAP (see also *DAP Series: FORTRAN-PLUS Language*).

Any module in the entire program can call any other module, with the following restrictions:

- Execution must begin and end in the host program
- A DAP module cannot call a host module
- Data can only be passed between the host program and the DAP program via APAL data sections with the COMMON property, or FORTRAN-PLUS COMMON blocks

9.1.2 Standard macro facilities

A suite of standard macros and data identities is available to simplify your task of creating code sections; in particular, of creating the interface between the code sections. You can use the macros and identities throughout a module if the APAL pre-processor file-include statement:

```
#include usrmacs.da
```

appears directly after the module header (see *DAP Series: Program Development*). This statement also includes a global data identity giving you access to a set of useful bit patterns, described in section 10.1. If you want to you can devise your own code section conventions rather than use the standard macros; guidelines for doing this are given in in section 9.6. Sections 9.2 to 9.5 assume that the standard suite is used. Details of the APAL instructions used in the examples appear in chapter 6 and appendix F.

9.2 Entry and exit conventions

This section describes the entry and exit conventions for an APAL code section and how it can access its parameters, if any.

9.2.1 Entry conventions

You should call the standard macro 'PROLOGUE immediately after every code header and every code section entry point. It takes one parameter, which is the number of parameters passed to the code section or entry point, the default being zero.

Example

```
CODE EXAMPLE DAP
'PROLOGUE 3                ! EXAMPLE HAS 3 PARAMETERS
.
.
.
ENTRY ENT1
'PROLOGUE 1                ! ENT1 HAS 1 PARAMETER
.
.
.
ENTRY ENT2 HOST
'PROLOGUE                  ! ENT2 MAY BE CALLED FROM
.                          ! THE HOST PROGRAM AND
.                          ! HAS NO PARAMETERS
.
END
```

9.2.2 Exit conventions

You should call the standard macro 'EPILOGUE to return to the calling routine. The most convenient place for it is just before the code section END statement. In cases where a return is required at several points in a code section, the macro call can be labelled and then jumped to using the J instruction. Alternatively, 'EPILOGUE can be called anywhere in the code section, but as it expands to several instructions it cannot be conditionally jumped around using SKIP.

9.2.3 Parameter access

When a code section is called, register M6 contains the address of its *name space*. This area of array store contains linkage information and the addresses of any parameters for the code section. You can access the parameters using the data identity 'PARBASE, which is the word offset from the address in M6 of a consecutive set of rows containing the addresses of the parameters.

Each parameter address occupies one word, the first such address being 'PARBASE..1 (M6), the second in 'PARBASE..2 (M6) and so on. If the code section is expected to return a result (thus behaving like a FORTRAN-PLUS FUNCTION), then the address of the result will be in the first word of the set, 'PARBASE (M6).

Some addresses are passed directly in registers to the code section, as well as being copied to the name space. A result address is passed in M1, the first parameter address in M2, the second in M3, and the last parameter address (if there are three or more) in M4.

Example

This example shows a FORTRAN-PLUS subprogram calling an APAL code section as though it were a FORTRAN-PLUS FUNCTION.

FORTRAN-PLUS subprogram

```
SUBROUTINE CALLSAPAL
EXTERNAL INTEGER SCALAR FUNCTION DIFF
.
.
.
J = DIFF (73, 31)
.
.
.
```

APAL code section

```
MODULE EXAMPLE2
#include usrmacs.da
.
.
.
```

```
.
.
.
CODE DIFF DAP
'PROLOGUE 2                  ! DIFF HAS 2 PARAMETERS
RW M1 'PARBASE (M6)          ! THE FUNCTION RESULT ADDRESS
RW M2 'PARBASE..1 (M6)       ! THE FIRST PARAMETER ADDRESS
RW M2 0 (M2)                 ! THE FIRST PARAMETER
RW M3 'PARBASE..2 (M6)       ! THE SECOND PARAMETER ADDRESS
RW M3 0 (M3)                 ! THE SECOND PARAMETER
SUB M2 M3                    !
WR M2 0 (M1)                 ! COPY THE ANSWER TO THE RESULT
.
.
' EPILOGUE
END
.
.
.
ENDMODULE
```

In this simple example M1, M2 and M3 could have been used directly instead of being loaded from the name space, thus saving three instructions.

9.3 Areas of the DAP available to a code section

This section describes the locations in a DAP whose contents can be altered by a code section.

9.3.1 Free name space

After a call of 'PROLOGUE, register M7 contains the address of the *free name space*. This is the first free plane in the name space of the code section that does not contain linkage information or parameter addresses. All planes in array store from this location up to the DAP Program Block limit are available to the code section. The size of this stack area can be adjusted at assembly time using the STACK statement (see section 10.4).

9.3.2 Workspace

Array store planes 0 to 119 in the DAP Program Block are freely available as workspace to any code section. Their contents are undefined on entry to the code section.

9.3.3 Other array store locations

As well as workspace and the free name space, the following array store locations can be accessed by a code section:

- The locations containing parameters passed by the calling routine
- The location containing the result to be passed back to the calling routine
- Data sections that are either local to the module containing the code section, or that have the DAP property

9.3.4 PE register planes, MCU registers and the edge register

The PE A, Q and C planes are freely available to a code section. Their contents are undefined on entry and need not be restored on exit.

The MCU registers M0 – M13 and the edge register ME are also available to a code section. Some registers contain useful information on entry or after a call of 'PROLOGUE. However you do not need to restore any of them on exit. It is convenient to maintain the value of M7 (the free name space address) throughout a code section.

9.4 Calling another code section

This section describes the conventions you should follow in an APAL code section when calling another APAL code section or FORTRAN-PLUS subprogram.

9.4.1 Creating a name space

A calling routine must create the name space for the called routine, and place its address in register M5. The only information that needs to be put into the name space is the result and parameter addresses; all other linkage information is created automatically by the standard macros. The address of the name space should be that of the first unused plane in the calling routine's free name space.

Example

```
RAX M5 97 (M7)
```

In this example the new name space begins 97 array store planes beyond the start of the current free name space.

The result and parameter addresses must now go in the new name space using the data identity 'PARBASE. Some addresses must also be made available directly in MCU registers (see section 9.2.3). All these addresses are supplied, in the most general case, as follows:

```
RAW    M1    function result address
WR     M1    'PARBASE (M5)
RAW    M2    parameter 1 address
WR     M2    'PARBASE..1 (M5)
RAW    M3    parameter 2 address
WR     M3    'PARBASE..2 (M5)
RA     M4    parameter 3 address
WR     M4    'PARBASE..3 (M5)
RAW    M4    parameter 4 address
WR     M4    'PARBASE..4 (M5)
.
.
.
RAW    M4    parameter n address
WR     M4    'PARBASE..n (M5)
```

You sometimes need to use an instruction other than RAW to load a parameter address into a register. When you are calling a FORTRAN-PLUS sub-program, do not put parameters in workspace, since a called routine can corrupt workspace before accessing its parameters.

9.4.2 Calling the routine by name

Once the name space has been created for the routine, you can then call it using the standard macro 'CALLNAME, which takes one parameter, the name of the routine to be called. On exit from the called routine, registers M6 and M7 hold the values of the calling routine's name space and free name space addresses. This is the same as after calling 'PROLOGUE on entry.

Example

```
RAX    M5    24 (M7)               ! NEW NAME SPACE
RAX    M2    10.3 (M7)             ! ONE PARAMETER AT PLANE 10 ROW 3
WR     M2    'PARBASE..1 (M5)      ! OF CURRENT FREE NAME SPACE
'CALLNAME    EX1                   ! CALL CODE SECTION EX1
```

9.4.3 Calling the routine by address

A routine can be called using its address instead of its name. Usually this is done when the name is unknown because the address of the routine was passed as a parameter to the calling code section; such calls are therefore referred to as *parametric* calls.

The address of the routine must be placed in M7. The routine is then called using the standard macro 'CALLPARAM, which has no parameters. As with 'CALLNAME, the name space and free name space addresses are restored to M6 and M7 on exit.

Example

```
RAX    M5     10 (M7)              ! NEW NAME SPACE.
RAX    M1     RESULT (M7)          ! CALLED ROUTINE RETURNS A
WR     M1     'PARBASE (M5)        ! RESULT. THE CODE SECTION
RW     M7     'PARBASE..4 (M6)     ! ADDRESS WAS PARAMETER 4 OF
                                   ! THE CALLING ROUTINE.
'CALLPARAM                         ! CALL CODE SECTION
```

9.4.4 Complete example

This example illustrates the above points. It is a complete example of one routine calling another, and is based on the example in section 9.2.3 except that the calling routine is now an APAL code section as well.

Calling module

```
MODULE CALLER
#include usrmacs.da
DATA ARTHUR HOST COMMON WRITE       ! A DATA SECTION WITH ONE
ANSWER: ROW                         ! UNINITIALISED ROW
END
CODE DENT HOST
'PROLOGUE                           ! DENT MAY BE CALLED FROM THE
                                    ! HOST SECTION AND SO HAS NO
                                    ! PARAMETERS
PAR1 = 0..0                         ! LOCAL DATA IDENTITIES
PAR2 = 0..1                         ! TO SIMPLIFY ACCESS TO
FREE = 1..0                         ! FREE NAME SPACE
RLIT   M1    73
WR     M1    PAR1 (M7)              ! PAR1 (M7) HOLDS 73
RLIT   M1    31
WR     M1    PAR2 (M7)              ! PAR2 (M7) HOLDS 31
RAW    M5    FREE (M7)              ! NEW NAME SPACE
RAR    M1    ANSWER                 ! RESULT ADDRESS IS IN
WR     M1    'PARBASE (M5)          ! DATA SECTION ARTHUR
RAW    M2    PAR1 (M7)
WR     M2    'PARBASE..1 (M5)       ! PARAMETER 1
RAW    M3    PAR2 (M7)
WR     M3    'PARBASE..2 (M5)       ! PARAMETER 2
'CALLNAME    DIFF                   ! CALL CODE SECTION DIFF
'EPILOGUE
END
ENDMODULE
```

Called module

```
MODULE CALLED
#include usrmacs.da
CODE DIFF DAP
'PROLOGUE 2                              ! DIFF HAS 2 PARAMETERS
RW    M1    'PARBASE (M6)                ! THE FUNCTION RESULT ADDRESS
RW    M2    'PARBASE..1 (M6)
RW    M2    0 (M2)                       ! PARAMETER 1
RW    M3    'PARBASE..2 (M6)
RW    M3    0(M3)                        ! PARAMETER 2
SUB   M2    M3                           ! M2 = M2 - M3
WR    M2    0 (M1)                       ! COPY ANSWER TO RESULT
'EPILOGUE
END
ENDMODULE
```

9.5 Calling an APAL code section from a host routine

A host routine can call an APAL code section. The calling process is the same as if the calling program were a FORTRAN-PLUS subroutine with the following differences:

- The called code section, or entry point, must have the HOST property rather than the DAP property
- No parameters can be passed from the host routine to the APAL code section. Similarly, the APAL code section cannot return a result to the host routine; thus it can not behave as a FORTRAN-PLUS function subprogram
- The transfer of control from the host program to the DAP program is performed via interface subroutines. These are described in *DAP Series: Program Development*

All communication between the host and APAL routines is via DATA sections.

9.5.1 Passing data between the host and APAL routines

Since a host routine is not permitted to pass parameters to or receive a result from an APAL routine, data is passed between the two via APAL DATA sections.

A DATA section used for this purpose is declared with the properties HOST, COMMON, and optionally WRITE.

Transfer of data to and from the APAL DATA section is made in units of store words and is carried out by interface routines called by the host program. How to use the interface subroutines is described in *DAP Series: Program Development*.

You should note the following points:

- Care should be taken to make sure that the data areas in both the host code and the DAP code (FORTRAN-PLUS or APAL) are large enough to accommodate the number of words being transmitted
- The sizes of APAL DATA sections are rounded up to multiples of a plane ($ES^2/8$ bytes), and the sections are aligned to plane boundaries
- In general, data storage modes used by the host differ from those used by the DAP – see section 8.4.1 in *DAP Series: FORTRAN-PLUS Language*. You may find it more convenient therefore to enter the DAP at a FORTRAN-PLUS routine where the appropriate mode conversion routines are available, and then call your APAL routine from your FORTRAN-PLUS code

An example of a complete DAP program and associated host program is given in appendix C.

9.6 User-defined conventions

If you want to adopt your own conventions for parameter passing and name space chaining (instead of using standard macros), you should follow the following guidelines:

- M6 must contain the address of the current name space. This is passed in M5 in a call from FORTRAN-PLUS or a host program
- Word 0..0 (M6) must contain the address of the calling routine's name space. This is passed in M6 in a call from FORTRAN-PLUS or a host program
- Word 0..1 (M6) must contain the link address (code address of the next instruction to be obeyed on return to the caller). This is passed in M0 in a call from FORTRAN-PLUS or a host program
- No other words in plane 0 (M6) should be used. The first word that a routine can use for parameter passing and general stack purposes is 1.0.0 (M6)
- To call a routine by name, use the following code:
  ```
  JESL name
  NULL
  ```
- To call a routine parametrically, use the following code:
  ```
  RW  M7   word containing code section address
  JESL AMT5PARAM                    ! AN AMT-SUPPLIED SUBROUTINE
  ```
- Before returning (using EXIT), M6 must contain the caller's name space address, and M0 the link address
- The run-time diagnostic system will give spurious information if an error occurs and a FORTRAN-PLUS routine has been entered but control has not been returned to its caller

Chapter 10

Miscellaneous facilities

This chapter describes a number of facilities not covered elsewhere. These facilities are:

- Accessing standard bit patterns
- Including source input from alternative files
- Requesting stack space
- Controlling the format of the assembler source output listing
- Generating messages to the assembly output listing file

10.1 Accessing standard bit patterns

You can make a set of useful bit patterns available to all code sections in a module by putting the file-include statement:

```
#include usrmacs.da
```

at the beginning of the module, after the module header (see *DAP Series: Program Development*). This statement also gives you access to the standard macros suite for code section conventions, as described in chapter 9. The *usrmacs.da* file includes global data identities associating standard identifiers with data addresses. Most of the patterns occupy a whole store plane but you can usefully access some of them on a row basis. As they are all in the same data section the easiest way of addressing them is by loading the start of section address into a register.

Example

```
    RASC M1     'BINARY_CHOP            ! LOAD M1 WITH ADDRESS OF
                                        ! PATTERNS DATA SECTION
    RX   ME     'BINARY_CHOP.1(M1)      ! LOAD ME WITH SECOND ROW
                                        ! OF 'BINARY_CHOP
    QS   'UNIT_DIAG(M1)                 ! LOAD Q PLANE WITH 'UNIT_DIAG
    .
    .
    .
```

The following sections illustrates the bit patterns corresponding to each identifier; all identifiers except 'UD_BASE address the start of a plane.

In each illustration, the most significant bit in each row is on the left of the row, the least significant bit on the right. Most illustrations show only the first few and the last few rows in the plane, and only the first 8 bits and the last 8 bits in the row; one illustration shows the middle 8 bits as well.

10.1.1 'BINARY_CHOP

The plane starts with a set of n rows with bits set much as shown below, where $n = \log_2 ES$, ES being the edge size of the target DAP on which the APAL code will be run; other rows in the plane are undefined. Hence for DAP 500 the first 5 rows are defined, for DAP 600 the first 6 rows are defined, and so on.

For DAP 600 the defined rows would be:

```
1111 1111 1111 1111 1111 1111 1111 1111 0000 0000 0000 0000 0000 0000 0000 0000
1111 1111 1111 1111 0000 0000 0000 0000 1111 1111 1111 1111 0000 0000 0000 0000
1111 1111 0000 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111 0000 0000
1111 0000 1111 0000 1111 0000 1111 0000 1111 0000 1111 0000 1111 0000 1111 0000
1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
1010 1010 1010 1010 1010 1010 1010 1010 1010 1010 1010 1010 1010 1010 1010 1010
```

For DAPs with a different ES a similar set of defined ($\log_2 ES$) rows are produced, each row being ES bits wide.

10.1.2 'UD_BASE

This name does not identify a pattern, but points to the row preceding 'UNIT_DIAG. The name is included so that you can index the individual rows of 'UNIT_DIAG from 1 instead of 0 if you need to.

10.1.3 'UNIT_DIAG

The plane identified by 'UNIT_DIAG contains a unit diagonal matrix. The ES rows are:

```
1000...  ...0000          row 0
0100...  ...0000          row 1
0010...  ...0000          row 2
0001...  ...0000          row 3
.
.
.
0000...  ...1000          row ES − 4
0000...  ...0100          row ES − 3
0000...  ...0010          row ES − 2
0000...  ...0001          row ES − 1
```

10.1.4 'TRAIL_DIAG

The plane identified by 'TRAIL_DIAG contains a trailing diagonal matrix. The *ES* rows are:

```
0000...  ...0001          row 0
0000...  ...0010          row 1
0000...  ...0100          row 2
0000...  ...1000          row 2
.
.
.
0001...  ...0000          row ES − 4
0010...  ...0000          row ES − 3
0100...  ...0000          row ES − 2
1000...  ...0000          row ES − 1
```

10.1.5 'ALTERNATE

Each row of this plane contains a repeating pattern of *n* false and *n* true bits where *n* is the row index numbering from 0. The *ES* rows are:

```
0000 0000 ...  0000 0000     all bits false
0101 0101 ...  0101 0101     false, true, false, true, ...
0011 0011 ...  0011 0011     two false bits , two true bits , ...
0001 1100 ...  .... ....     three false bits , three true bits , ...
.
.
.
0000 0000 ...  0000 0111     first ES − 3 bits false, last three bits true
0000 0000 ...  0000 0011     first ES − 2 bits false, last two bits true
0000 0000 ...  0000 0001     first ES − 1 bits false, last bit true
```

10.1.6 'LOWER_TRI

This plane contains a lower triangular matrix, the *ES* rows being:

```
1000 0000 ...  0000 0000            row 0
1100 0000 ...  0000 0000            row 1
1110 0000 ...  0000 0000            row 2
1111 0000 ...  0000 0000            row 3
1111 1000 ...  0000 0000            row 4
.
.
.
1111 1111 ...  1111 1100            row ES - 3
1111 1111 ...  1111 1110            row ES - 2
1111 1111 ...  1111 1111            row ES - 1
```

10.1.7 'SHUFFLE

This plane contains a pattern useful in perfect shuffle operations, the *ES* rows being:

```
bit 0                  bit ES/2            bit ES - 1
↓                      ↓                   ↓

1000 0000 ...  0000 0000 ...  0000 0000        row 0
0000 0000 ...  0000 1000 ...  0000 0000        row 1
0100 0000 ...  0000 0000 ...  0000 0000        row 2
0000 0000 ...  0000 0100 ...  0000 0000        row 3
0010 0000 ...  0000 0000 ...  0000 0000        row 4
.
.
.
0000 0000 ...  0010 0000 ...  0000 0000        row ES - 4
0000 0000 ...  0000 0000 ...  0000 0010        row ES - 3
0000 0000 ...  0001 0000 ...  0000 0000        row ES - 2
0000 0000 ...  0000 0000 ...  0000 0001        row ES - 1
```

10.1.8 'UNSHUFFLE

This plane contains a pattern for selecting alternate bits in sequence.

```
1000 0000 ...  0000 0000            row 0
0010 0000 ...  0000 0000            row 1
0000 1000 ...  0000 0000            row 2
0000 0010 ...  0000 0000            row 3
.
.
```

```
0000 0000 ...  0000 1000            row ES/2-2
0000 0000 ...  0000 0010            row ES/2-1

0100 0000 ...  0000 0000            row ES/2
0001 0000 ...  0000 0000            row ES/2+1
0000 0100 ...  0000 0000            row ES/2+2
0000 0001 ...  0000 0000            row ES/2+3
.
.
0000 0000 ...  0000 0100            row ES - 2
0000 0000 ...  0000 0001            row ES - 1
```

10.2 Incorporating source from alternative files

You can specify that the assembler is to read source from another file, by using the APAL preprocessor file-include facility (see *DAP Series: Program Development*). At the end of the included file, assembly continues at the statement after the file-include statement in the original file. File-include statements can be nested.

10.3 Requesting stack space

You can request stack space at assembly time using the STACK statement.

10.3.1 Syntax

<STACK statement> ::= STACK<numval><new line>

10.3.2 Semantics

STACK can appear anywhere within an APAL module, and has the form:

STACK *plane-count*

where *plane-count* is an unsigned integer or hexadecimal value, or an assembly-time expression enclosed in [ ] yielding a non-negative integer value.

plane-count is the size of the stack space request in store planes. Several STACKs can appear in the module; the assembler takes the largest value of *plane-count* for each module.

The consolidator creates a stack section in the DOF file which is the sum of the stack space requests from all the included CIF modules. There are ways of increasing this sum or ignoring it altogether and using an absolute value at consolidation time; for more details see *DAP Series: Program Development*.

When you are calculating the value to be used in a STACK statement, allow for the difference of a few planes between the name space and the free name space of a code section; see chapter 9 for more details.

10.4 Controlling the output listing

Two assembly-time facilities control the output listing produced by the assembler. You have access to these facilities through the APAL statements:

    LIST
    NOTE

10.4.1 The LIST statement

The LIST statement alters the current listing option for the assembler output listing.

10.4.1.1 Syntax

<LIST statement> ::= LIST<list option><new line>

<list option> ::= FULL | SOURCE | SHORT | NONE

10.4.1.2 Semantics

Multiple occurences of LIST are allowed; a statement can appear anywhere within an APAL module, over-riding any LIST option already in force The statement has the form:

LIST *list-option*

where *list-option* can be FULL, SOURCE, SHORT, or NONE.

The options FULL, SOURCE and SHORT produce the same effect as the parameters 3, 2, and 1 respectively to the *L* flag for the **dapa** command when the APAL source code is assembled; the NONE option for the LIST statement results in no listing, and is equivalent to specifying no **dapa** **L** flag. If a **dapa** **L** flag is specified, and LIST statement(s) are used in the APAL source code, then the **L** flag only stays in force until the first LIST statement is encountered. The listing option controls the type and amount of information produced in the assembler output listing; the detailed significance of the values of *list-option* are described in *DAP Series: Program Development*.

If LIST appears in a macro body, the specified option remains in force until exit from the macro or until it is changed in a subsequent LIST in the same, or nested, macro body. On exit from the macro the listing option is reset to its value at the time of the macro call.

LIST itself will only appear in the assembler output listing if the listing option in force when LIST is encountered is such that assembly-time statements are listed.

10.4.2 The NOTE statement

The NOTE statement outputs a message to the assembler output listing.

10.4.2.1 Syntax

<NOTE statement> ::= NOTE<note type><string><new line>

<note type> ::= TERMINAL | ERROR | WARNING | COMMENT

10.4.2.2 Semantics

NOTE can appear anywhere within an APAL module, and has the form:

> NOTE *note-type string*

where *note-type* can be TERMINAL, ERROR, WARNING, or COMMENT.

NOTE allows you to output a message to the assembler output listing file. *note-type* specifies the type of message and its format; *string* constitutes the text of the message.

Messages output by NOTE are treated in the same way as messages generated by the assembler (see *DAP Series: Program Development*). The assembler message count is incremented each time NOTE generates a message. If a TERMINAL message is generated, assembly is abandoned.

The NOTE statement is generally only useful when you are using conditional assembly (see chapter 11 for more details).

Chapter 11

Substitutions and conditional assembly

The APAL language allows you to define and manipulate values of *assembly-time variables* (see section 11.2). The *assembly-time values* (see section 11.1) held in these variables are character strings, or arithmetic expressions whose numerical results at assembly time are then regarded as character strings. As part of the assembly process you can specify that the current value of an assembly-time variable is to be substituted into a specified point in your APAL code text prior to the assembler analysing it as an APAL statement (see section 11.3 for details).

There is a closely related set of facilities that allows you to manipulate and use macro parameters and macro variables; chapter 12 describes facilities specific to macros.

You can control the assembly process further, by specifying conditions under which the assembler is to ignore specified sequences of statements (see section 11.4 for details); these statement-ignoring conditions typically are the result of a comparison between assembly-time values.

You may sometimes want to include in your character strings some special characters (such as a '!', which normlly marks the start of a comment). You use the escape character mechanism to include these characters; you will find many references to it, both in this chapter and in chapter 12. However, you will find in practice that you will not often want to use special characters, so on a first reading of these chapters you can safely ignore all references to the escape character (^), and to the exact meaning of and distiction between protected and unprotected strings.

11.1 Assembly-time values

assembly-time values are character strings, or expressions which are evaluted at assembly time and the results regarded as character strings.

11.1.1 Syntax

<assembly-time value> ::= [<assembly-time expression>] | <string>

<string> ::= <string character>*<delimiter>

<string character> ::= ^ <character> | ^ | " | <basic character>

<delimiter> ::= , |] | ! | <newline>

<assembly-time expression> ::= <sign>?<expression>

<expression> ::= <expression><operator><operand> | <operand>

```
<operand> ::= <assembly-time variable name> |
              <macro variable name> |
              <macro parameter name> |
              <unsigned integer> |
              <hexadecimal value> |
              (<assembly-time expression>)
```

<operator> ::= + | - | <star> | / | // | 'AND | 'NEQ | 'OR | 'ROTATE | 'SCALE

11.1.2 Semantics

An assembly-time value can be either a *string* or an *assembly-time-expression* enclosed in [ ]. If the first character of an assembly-time value is [, it is treated as an assembly-time expression, otherwise it is treated as a string.

11.1.2.1 Strings

A string is a sequence of characters (or a null string) as defined in section 11.1.1, and is terminated by a delimiter. The type of delimiter that can terminate a string depends on the context in which the string appears.

When a string is encountered by the assembler it is extracted from the source text as follows:

- All underscore characters and leading spaces are removed
- Each pair of characters of the form:
  ^*character*
  is copied to the string, regardless of the value of *character*
- All other characters, except for underscore characters, are copied to the string, up to, but not including, the string delimiter appropriate to that context
- <newline> always acts as a string delimiter. The ! character is also a string delimiter unless preceded by the escape character ^, in which case it is a *protected character* and is copied to the string. Comma and] only delimit strings in certain contexts that are described later in this chapter; they can be included in a string as protected characters if they are preceded by ^

- Trailing spaces and underscore characters are normally ignored. Note however that if a string terminates with a ^ character that is immediately followed by space or underscore, the space/underscore is transferred to the string together with the ^ character

Note that the significant part of a string (that part that remains after leading and trailing spaces and underscores have been removed) must not be longer than one line (80 characters).

A string that is extracted from the text in this way is called a *protected string*, because it can contain special characters *protected* by the escape character ^.

However, in some cases, described later in this chapter, it is necessary to remove ^ characters to form an *unprotected string*, which is derived from a protected string as follows:

- Each character of the protected string is copied to the unprotected string until a ^ character is encountered
- If ^ is not the last character in the protected string, it is ignored and the following character, whatever it may be, is copied to the unprotected string. If ^ is the last character in the protected string, it is copied to the unprotected string

A protected string can be changed into an unprotected string explicitly by variable substitutions (see section 11.4.1).

The assembler will also use the unprotected form of a string in certain assembly-time operations (see sections 11.4.7, 11.4.8 and 12.2), although the string itself will not be altered.

Examples

```
A PROTECTED STRING CONTAINING A ^! CHARACTER
THIS STRING TERMINATES WITH ONE SPACE ^
```

Note that if ^ is not immediately followed by <newline>, the ^ and the first of any following space character(s) are included in the string.

11.1.2.2 Assembly-time expressions

An assembly-time expression is a combination of *operands* and *operators* that can be evaluated at assembly-time to produce an assembly-time value. An *operand* is an unsigned integer, hexadecimal value, or assembly-time variable, macro variable, or macro parameter whose value is an optionally signed basic integer; an *operator* defines the function to be performed on operands, and can be any of the following:

Operator	*Effect*
+	Addition
–	Subtraction
*	Multiplication

Operator	*Effect*
/	Integer division. The sign of the result is the same as that of the dividend; the remainder is ignored
//	Integer division, giving the remainder. The remainder has the same sign as the dividend; the quotient is ignored
'AND	The logical AND of the two operands, each treated as unsigned 32-bit integers
'OR	The logical OR of the two operands, each treated as unsigned 32-bit integers
'NEQ	The logical non-equivalence of the two operands, each treated as unsigned 32-bit integers
'ROTATE	The first operand is treated as an unsigned 32-bit integer, and shifted left cyclically through the number of bit positions given by the second operand. A negative second operand gives a right shift
'SCALE	The first operand is treated as an unsigned 32-bit integer, and shifted left the number of bit positions given by the second operand. A negative second operand gives a right shift. In either case zero bits are fed into the 'empty' bit positions

The order of evaluation is such that the operators *, /, and // take precedence over +, –, 'AND, 'OR, 'NEQ, 'ROTATE and 'SCALE. The order of evaluation for operators of equal precedence is from left to right; you can use parentheses to alter the order of evaluation.

For example:

A + B * C is the same as A + (B * C)
A/B * C is the same as (A/B) * C

The unary operators + or – must be either the first symbol in an expression or the symbol immediately following an open parenthesis.

For example:

–A + B * (–C/D)

Assembly-time expressions are evaluated to 32-bit precision. If overflow occurs, an error is flagged by the assembler.

11.2 Assembly-time variables

An assembly-time variable is a data item that is processed by the assembly-time facilities described in this chapter. Once you have declared an assembly-time variable, you can use its value in an assembly-time expression or substitute the value into the source text (see section 11.4).

You declare an assembly-time variable (and optionally give it an initial value) by using the assembly-time statement VAR (see section 11.2.1); you can give a declared assembly-time variable a new value by means of the assembly-time statement SET (see section 11.2.2). Assembly-time variables can either be declared or redefined or both, either within or outside a module.

The value assigned to an assembly-time variable is a character string: that value can either be a normal alphanumeric character string or the character representation of an integer value. You can use this value in the source program either by writing the name of the variable as an operand of an assembly-time expression or by substituting the value itself into the source text (see section 11.4.1).

11.2.1 The VAR statement

The VAR statement declares and optionally gives an initial value to an assembly-time variable.

11.2.1.1 Syntax

<VAR statement> ::= VAR<assembly-time variable name><preset>?<newline>

<assembly-time variable name> ::= <identifier>

<preset> ::= =<assembly-time-value>

11.2.1.2 Semantics

Before you can reference an assembly-time variable you must declare it in a VAR statement, which has the form:

VAR *name* = *assembly-time-value*

where

name is an identifier whose scope is determined by the context of VAR as follows:

- If VAR appears outside a module, *name* has external scope and must be different from the names of all other assembly-time variables
- If VAR appears within a module, *name* has global scope within that module and must be different from the names of all the other assembly-time variables you declare within the module and from any assembly-time variables you have declared previously and whose names have external scope

= *assembly-time-value* is optional, and allows you to specify an initial value for the variable. This value must be a string or an assembly-time expression enclosed in [ ], otherwise the initial value of the variable is the null string.

If *assembly-time-value* is a string, it is delimited either by <newline> or by the first unprotected ! character (see section 11.1.1).

Examples

```
VAR A = THIS IS A STRING 35 CHARACTERS LONG
VAR B = -11                 ! THIS VALUE MAY BE USED IN AN ASSEMBLY-TIME
                            ! EXPRESSION, OR MAY BE SUBSTITUTED INTO THE
                            ! PROGRAM SOURCE AS THE THREE CHARACTERS
                            ! -11
VAR C = [B+1]               ! C HAS INITIAL VALUE -10.
VAR A = 6                   ! THIS IS INVALID, SINCE A HAS ALREADY BEEN
                            ! DECLARED.
VAR D = [THIS IS INVALID; IT IS NOT AN ASSEMBLY-TIME EXPRESSION]
VAR E = 6+B*3               ! THIS IS A STRING, NOT AN EXPRESSION (THERE
                            ! ARE NO SURROUNDING [ ]).
```

11.2.2 The SET statement

The SET statement assigns a value to an assembly-time variable you have declared previously.

11.2.2.1 Syntax

<SET statement> ::= SET<assembly-time variable name><preset><newline>

11.2.2.2 Semantics

You can *redefine* an assembly-time variable, that is, give it a new assembly-time value, by using SET, which has the form:

SET *name* = *assembly-time-value*

where

name is the name of an assembly-time variable that is currently in scope

assembly-time-value is the assembly-time value you want to assign to *name*, and can be a null string

11.3 Substitutions

With the exceptions listed below, each input line, including comments, is scanned at assembly time for any *textual substitutions* you require; these substitutions show what part of the text of a line you want to replace by alternative text, for example, by the character value of some assembly-time variable.

You use the % character to show the start of such a substitution.

The exceptions to this generalisation are:

- The lines of a macro body during the assembly of the macro definition; that is, all lines following a MACRO statement up to and including the corresponding MEND (see Chapter 12). These lines will be scanned for substitutions if and when the macro is called
- Any lines that are omitted in a conditional assembly (for details, see section 11.4, and the CYCLE construct in section 12.6)
- A % character that is preceded by an odd number of ^ characters. The ^(s) prevents any substitution from taking place, and the text is unchanged

Lines that are scanned for substitutions before being interpreted as instructions need not be syntactically correct until all substitutions have been performed.

Like most computer languages APAL has a macro facility, and chapter 12 discusses macros in detail. All of the facilities discussed in this section are available in macros, a few are available only in macros. To make formal description of these facilities complete, substitutions only available in macros are included in the syntax statements in this section, and at a few other points. So as not to complicate this section unnecessarily, no discussion is offered on those macro facilities; they are marked by comments such as:

! *only in macros, see chapter 12*

These macro facilities are fully discussed in chapter 12.

11.3.1 Syntax

```
<substitution> ::= <variable substitution> |
                   <expression substitution> |
                   <section substitution> |
                   <plane substitution> |
                   <row substitution> |
                   <column substitution> |
                   <substring substitution> |
                   <length substitution> |
                   <repeated substitution> |
                   <parameter substitution>        ! only in macros, see chapter 12
```

If the text immediately following a % character is not a valid substitution, an error is flagged by the assembler.

11.3.2 Variable substitutions

In a variable substitution, an assembly-time variable, macro variable (see chapter 12 for more details) or system variable in a line of code is replaced by the character string representing the assembly-time value of that variable.

11.3.2.1 Syntax

```
<variable substitution> ::= %<assembly-time variable name> |
                            %^ <assembly-time variable name> |
                            %<system variable name> |
                            %<macro variable name> |      ! only in macros, see chapter 12
                            %^ <macro variable name>      ! only in macros, see chapter 12

<system variable name> ::= 'DATE | 'DIM | 'LOGDIM | 'TIME | 'TRANSFER |
                           'MCOUNT |                      ! macro facility, see chapter 12
                           'PCOUNT                        ! only in macros, see chapter 12
```

11.3.2.2 Semantics

A variable substitution causes the name of an assembly-time, macro, or system variable and the preceding % or %^ character to be replaced at assembly-time by the character value of the variable. If the substitution refers to a variable whose name is not in scope, an error is flagged by the assembler. See chapter 12 for details of macro variables.

A variable substitution can have any of the following forms:

- *%assembly-time-variable-name*, which is replaced by the value of the named assembly-time variable
- *%^assembly-time-variable-name* which is the same as *%assembly-time-variable-name* except that all pairs of characters in the value of the form:

  ^character

  are replaced by *character* alone; that is, the value of the assembly-time variable is converted to its unprotected form (see section 11.1.2.1)
- *%macro-variable-name* and *%^macro-variable-name*, discussed in chapter 12
- *%system-variable-name*, which is replaced by the value of the named *system variable*

  The following system variables can be referenced in a variable substitution:

System variable name	*Value*
'DIM	This variable is replaced by a character string representing the value of ES, the dimension of the DAP array for which the assembly is being carried out. For example, for DAP 600, 'DIM is replaced by the two characters '64'
'LOGDIM	This variable is replaced by a character string representing the value of $\log_2 ES$; for example, again for DAP 600, 'LOGDIM is replaced by the character '6'
'MCOUNT	See chapter 12 for details
'PCOUNT	See chapter 12 for details

System variable name	*Value*
'DATE	This variable is replaced by characters representing the value of the date on which the assembly began, in the form: *YYYY/MM/DD* For example, 'DATE might be replaced by the characters '1988/7/30'
'TIME	This variable is replaced by characters representing the value of the time (expressed in 24-hour clock time) at which the assembly began, in the form: *HH:MM:SS* For example, 'TIME might be replaced by the characters '09:32:45'
'TRANSFER	This variable is replaced by characters that represent the value of the 32-bit word offset, from the beginning of the current code section, of the next instruction to be generated in the code section. If this substitution appears outside a code section, a warning is output and the character '0' is substituted

Only the ' character and the following two characters of the system variable name need be specified

11.3.3 Expression substitutions

In an expression substitution, an assembly-time expression is replaced by the value of that expression.

11.3.3.1 Syntax

<expression substitution> ::= %=[<assembly-time expression>]

11.3.3.2 Semantics

The value of an assembly-time expression can be substituted for that expression at assembly-time. The expression must be as defined in section 11.1.

For example, if in the line:

```
DATA1: %=[3 + A]
```

A is an assembly-time variable with an assembly-time value of 7, then at assembly-time the expression [3 + A] is replaced by 10 and the line becomes:

```
DATA1: 10
```

11.3.4 Section substitutions

In a section substitution, the relevant text is replaced at assembly-time by the name of the section associated with a data address identified in that text (see section 7.3.3).

11.3.4.1 Syntax

<section substitution> ::= %'SECTION[<data address>]

11.3.4.2 Semantics

A section substitution has the form:

%'SECTION[*data address*]

In a section substitution, %'SECTION[*data address*] is replaced at assembly-time by the name of the section of store associated with the address. If *data address* specifies an absolute address (that is, there is no section name), a null string is substituted.

The keyword 'SECTION can be abbreviated to 'SE.

For example, in code to run on a DAP 600, given:

```
DATA MAT
.
.
.
END
DEFINE

DOG = MAT+2.9.1
CAT = MAT+3.27.0

END
```

then both the text:

```
%'SECTION[DOG]
```

and the text:

```
%'SECTION[CAT]
```

would be replaced at assembly-time by the character string MAT.

11.3.5 Plane substitutions

In a plane substitution the relevant text is replaced at assembly-time by the plane offset associated with the data address identified in that text.

11.3.5.1 Syntax

<plane substitution> ::= %'PLANE[<data address>]

11.3.5.2 Semantics

A plane substitution has the form:

> %'PLANE[*data address*]

In a plane substitution, %'PLANE[*data address*] is replaced at assembly-time by the value of the plane offset from the start of the section of store in which the address occurs. If the address is an absolute address, the plane offset is relative to the start of the array store part of the DAP program block.

The keyword 'PLANE can be abbreviated to 'PL.

For example, using the example in the previous section:

> %'PLANE[DOG]

is replaced at assembly-time by 2.

11.3.6 Row substitutions

In a row substitution, the relevant text is replaced at assembly-time by the row offset from the start of plane of the data address.

11.3.6.1 Syntax

<row substitution> ::= %'ROW[<data address>]

11.3.6.2 Semantics

A row substitution has the form:

> %'ROW[*data address*]

In a row substitution, %'ROW[*data address*] is replaced at assembly-time by the value of the row offset from the start of the plane associated with [*data address*.

The keyword 'ROW can be abbreviated to 'RO.

For example, using the example in section 11.4.4:

> %'ROW[CAT]

is replaced at assembly-time by 27.

11.3.7 Column substitutions

In a column substitution, the relevant text is replaced at assembly-time by the column offset of the column identified in that text.

11.3.7.1 Syntax

<column substitution> ::= %'COLUMN[<data address>]

11.3.7.2 Semantics

A column substitution has the form:

%'COLUMN[*data address*]

In a column substitution, %'COLUMN[*data address*] is replaced at assembly-time by the value of the column offset from the start of the plane associated with the data address.

The keyword 'COLUMN can be abbreviated to 'CO.

For example, using the example in section 11.4.4:

%'COLUMN[DOG]

is replaced at assembly-time by 9.

11.3.8 Word substitutions

In a word substitution, the relevant text is replaced at assembly-time by the word offset from the start of a row of the word identified in that text.

On code to run on a DAP 500, words and rows are equivalent, so such a substitution always results in the value 0 being substituted for the text.

11.3.8.1 Syntax

<word substitution> ::= %'WORD[<data address>]

11.3.8.2 Semantics

A word substitution has the form:

%'WORD[*data address*]

In a word substitution, %'WORD[*data address*] is replaced at assembly-time by the value of the word offset from the start of the row associated with *data address*.

The keyword 'WORD can be abbreviated to 'WO.

For example, using the example in section 11.3.4:

```
    %'WORD [DOG]
```

is replaced at assembly time by 1.

11.3.9 Substring substitutions

In a substring substitution, the relevant text is replaced at assembly-time by a substring derived from the value of the variable or parameter quoted in that text.

11.3.9.1 Syntax

```
<substring substitution> ::= %'SUBSTRING[<selector>]<string ref>

<selector> ::= <number>?SIZE<number>? |
               FROM<string> |
               UPTO<string> |
               FROM<string>, UPTO<string>

<string ref> ::= <assembly-time variable name> |
                 <number> |                          ! only in macros, see chapter 12
                 <macro variable name> |             ! only in macros, see chapter 12
                 <macro parameter name>              ! only in macros, see chapter 12
```

11.3.9.2 Semantics

A substring substitution allows you to select a substring from:

- The value of an assembly-time variable
- The value of a macro variable or parameter – see chapter 12 for details

Note that the substring inserted into the line of code can contain leading or trailing spaces. These spaces will be removed in the usual way if the substring is later used as a string.

A substring substitution has the form:

%'SUBSTRING [*selector*] *name*

or, in the case of a positional macro parameter (see chapter 12 for details):

%'SUBSTRING [*selector*] *number*

where

name, if specified, is the name of an assembly-time variable that is in scope; otherwise an error is flagged by the assembler and no substitution is performed

number is discussed in chapter 12

selector specifies how the substring is to be selected from *name* and can take any of the following values:

- *number*$_1$ SIZE *number*$_2$

  number$_1$ specifies the character position (numbered from zero) at which the selected substring is to begin, and must therefore be less than the number of characters in the value of *name*. If *number*$_1$ is omitted, zero is assumed. If *number*$_1$ is greater than the number of characters in the value of *name*, a null string is substituted

  number$_2$ specifies the number of characters in the selected substring; for example:

  ```
  VAR A = ABCDEFGH

  %'SUBSTRING [2 SIZE 3] A    ! SELECTS THE SUBSTRING CDE
  ```

  If *number*$_2$ is omitted, or if it specifies more characters than are left in the value of *name*, all the remaining characters are selected.

  For example:

  ```
  %'SUBSTRING [2 SIZE] A         ! SELECTS THE SUBSTRING CDEFGH
  %'SUBSTRING [4 SIZE 6] A       ! SELECTS THE SUBSTRING EFGH
  ```

  If *number*$_2$ is zero the result of the substitution is a null string.

  The above description applies to values that contain no escape characters; the effect of the presence of escape characters is described in section 11.3.9.3

- FROM *string*$_1$

  string$_1$ is delimited by] or a comma. The selected substring is that string within the value of *name* that follows, but does not include, the first occurrence of *string*$_1$.

  For example:

  ```
  VAR J = A JACKSON IN YOUR HOUSE
  %'SUBSTRING [FROM IN]J           ! SELECTS THE SUBSTRING (WITH
                                   ! LEADING SPACE)  YOUR HOUSE

  %'SUBSTRING [FROM HOUSE]J        ! SELECTS THE NULL STRING
  ```

  If *string*$_1$ does not occur in the name, a null string is selected.

  The above description is valid if *name* and *string*$_1$ contain no escape characters.

Note that the ! character must not appear in *string*$_1$ unless protected by the ^ escape character, otherwise it would be interpreted as the start of a comment

- UPTO *string*$_1$

  string$_1$ is delimited by]. The selected substring is that string within the value of *name* that precedes, but does not include, the first occurrence of *string*$_1$.

  For example:

  ```
  VAR K = B FLAT MINOR SEVENTH

  %'SUBSTRING [UPTO MINOR]K        ! SELECTS THE SUBSTRING
                                   ! (WITH TRAILING SPACE) B FLAT
  ```

  If *string*$_1$ does not occur in the name, the entire string is selected.

  The above description is valid if *name* and *string*$_1$ contain no escape characters.

  Note that the ! character must not appear in *string*$_1$ unless protected by the ^ character, otherwise it would be interpreted as the start of a comment

- FROM *string*$_1$, UPTO *string*$_2$

  string$_1$ is delimited by the comma; *string*$_2$ is delimited by].

  The selected substring is that string within the value of *name* that:

  - Follows, but does not include, the first occurrence of *string*$_1$
  - Precedes, but does not include, the first occurrence of *string*$_2$ following *string*$_1$

  For example:

  ```
  VAR X = THE JOHN COLTRANE QUARTET

  %'SUBSTRING [FROM THE, UPTO QUARTET]X

                          ! SELECTS THE SUBSTRING
                          ! JOHN COLTRANE
                          ! (WITH A SPACE BEFORE AND AFTER)
  ```

  If *string*$_1$ is not within the value of *name*, a null string is selected; if *string*$_2$ is not within the value, the string following, but excluding the first occurrence of, *string*$_1$ is selected.

  The above description is valid if none of *name*, *string*$_1$, and *string*$_2$ contain escape characters.

  Note that the ! character must not appear in *string*$_1$ or *string*$_2$ unless protected by the ^ character, otherwise it would be interpreted as the start of a comment

The keyword 'SUBSTRING can be abbreviated to 'SU.

A similar form of substring substitution is used with macro parameters; see chapter 12 for details.

11.3.9.3 Escape characters in substring substitutions

You may like to omit this section on first reading of the manual, as it deals with the somewhat complex and infrequently-used escape character mechanism.

If the value of *name* or any selector strings in a substring substitution contains the escape character (^), the substring selection process is carried out as follows:

- The original string and any selector strings are converted to the unprotected form, as described in section 11.1.2.1
- The appropriate substring selection is then done, as described in section 11.4.9.2
- The resulting substring is converted back to a protected form by inserting the escape character immediately before any character where one was removed from the original string in the first step above

Examples

The examples below perform substring selection on an assembly-time variable declared as follows:

VAR A = AB^CD^^EF^^^GHJK^<space><newline>

Note that the space character before the end of the line is part of the string because an escape character preceded it.

The unprotected form of the string is ABCD^EF^GHJK<space>

- %'SU [3 SIZE 4] A

  Here the unprotected form D^EF is selected, and D^^EF is substituted
- %'SU [FROM ^D] A

  ^D in unprotected form is D, so ^EF^GHJK <space> is selected, and ^^EF^^^GHJK^<space> is substituted
- %'SU [UPTO G^HJ] A

  G^HJ in unprotected form is GHJ, so ABCD^EF^, is selected, and AB^CD^^EF^^ is substituted. Note that only two ^s appear at the end of this substituting string; the third ^ in the original string is associated with the G, so is not 'put back' in the substituting string
- %'SU [FROM A^B^C, UPTO H^JK^^] A

  A^B^C in unprotected form is ABC.
  However, H^JK^^ in unprotected form is HJK^, which does not occur in the unprotected form of A, so the selection goes up to the end of the string.

  Hence the selected unprotected string is D^EF^GHJK<space>, and D^^EF^^^GHJK^ <space> is substituted
- %'SU [FROM ABC, UPTO H^JK^] A

  This is a syntatic error. The escape character preceding] prevents it from being interpreted as the selector delimiter. It and the following A form part of the string H^JK^] A
- %'SU [FROM BC^D^^, UPTO H^JK^<space>] A

  BC^D^^ in unprotected form is BCD^.
  H^JK^<space> in unprotected form is HJK<space>.

  Hence the selected unprotected string is EF^G, and EF^^^G is substituted

11.3.10 Length substitutions

In a length substitution the relevant text in the code line is replaced at assembly-time by the number of characters in the value of the variable specified in that text.

11.3.10.1 Syntax

<length substitution> ::= %'LENGTH <string ref>

11.3.10.2 Semantics

In a length substitution the possible variables are:

- An assembly-time variable
- A macro variable or parameter. See chapter 12 for details

A length substitution has the form:

> %'LENGTH *name*
> %'LENGTH *number*

where

> *name* is the name of a previously declared assembly-time variable, macro variable or macro parameter
>
> *number* is discussed in chapter 12

The value substituted is the length of the unprotected form of the string associated with the named variable.

If *name* does not exist or has a null string value, zero is substituted.

The keyword 'LENGTH can be abbreviated to 'LE.

Examples

```
VAR A = 12497

VAR B = [2*A]

VAR C = 2*A

VAR D = AB^C^^DEF^G

VAR E = HORRIBLY COMPLICATED

VAR F = %'SU [UPTO COMP] E
VAR G
%'LE A      ! SUBSTITUTES TO 5
%'LE B      ! SUBSTITUTES TO 5 (B IS THE STRING 24994)
%'LE C      ! SUBSTITUTES TO 3
%'LE D      ! SUBSTITUTES TO 8 (THE UNPROTECTED EQUIVALENT OF D
            ! IS THE STRING ABC^DEFG)
%'LE F      ! SUBSTITUTES TO 9. NOTE THAT THE LENGTH INCLUDES THE
            ! SPACE PRECEDING COMP
%'LE G      ! SUBSTITUTES TO 0
```

11.3.11 Repeated substitutions

A repeated substitution allows you to nest a series of substitutions and control the order in which they are performed.

11.3.11.1 Syntax

<repeated substitution> ::= %<substitution>

11.3.11.2 Semantics

When a line is scanned for substitutions, all substitutions indicated by single % characters are performed in the order of their appearance. If a pair of consecutive unprotected % characters is encountered, it is replaced by a single % character, and scanning of the remainder of the line continues. On completion, the line is known to contain at least one % character and consequently the line is rescanned. This process is repeated until all substitutions have been performed.

For example, given:

```
VAR A = B
VAR B = 12
```

then:

```
%%%A
```

first substitutes to:

```
%B
```

which then substitutes to:

```
    12
```

A further example:

```
    %%=[%'ROW[A]-%'COLUMN[B]]
```

is replaced by the character string representing the difference between the row address of A and the column address of B.

11.3.12 Concatentation within substitutions

Whenever substitutions are performed, *concatenations* can also be performed. They can be either *implicit* or *explicit*.

11.3.12.1 Implicit concatenations

An implicit concatenation is performed whenever a substitution is performed; the string produced by the substitution is concatenated with the string that precedes the % character that caused the substitution.

For example:

```
    VAR COUNT = 0
    VAR S
    .
    .
    .
    IF NE %COUNT, 1
    SET S = S

    FI
    .
    .
    .
    ! %COUNT ERROR%S FOUND
```

When this code is assembled, an S will be appended to ERROR if COUNT is not 1, otherwise the null string is added. Depending on the value of COUNT when the comment line is reached in the assembly, the line might perhaps be printed out as:

```
    !  0 ERRORS FOUND
```

11.3.12.2 Explicit concatenations

An explicit concatenation is performed whenever a substitution is immediately followed by the & character, in which case the string following & is concatenated with the substituting string.

For example, given:

```
VAR A = ABC
```

then:

```
%A&DEF
```

is replaced by the string:

```
ABCDEF
```

Whenever the & character is found in this context, it is removed and two halves of the line are concatenated. If two or more adjacent & characters are found in this context, only one is removed per scan of the line. The presence of another & character does not itself cause the line to be re-scanned; & is only interpreted as a concatenation character if it appears immediately after a substitution.

Note that you generally only require explicit concatenation when you are using substitutions to create text in which spaces are significant; for example, identifiers, values, and labels. In other circumstances you will find that the implicit concatenation caused by a substitution is sufficient.

For example, given::

```
VAR A = B
VAR B = 17
```

then

```
%%%A&&4
```

substitutes first to:

```
%B&4
```

The repeated substitution causes the line to be re-scanned, thus producing the substitution:

```
174
```

11.4 Conditional assembly

In general, the assembler processes files of APAL source sequentially, assembling the entire contents of each file. However, by using the IF construct you can specify that a particular part of the APAL source is only to be assembled if a specified condition is satisfied. The IF construct allows you to specify this conditional assembly.

As will be discussed in chapter 12, if you want to achieve conditional assembly in a macro, you can also use the CYCLE contruct – see chapter 12 for details.

11.4.1 The IF construct

An IF construct allows you to specify that a certain part of the APAL source is only to be assembled if a specified condition is true.

11.4.1.1 Syntax

```
<IF construct> ::= <IF test><choice>FI<newline>

<IF test> ::= IF<condition> | IFN<condition>

<choice> ::= <true part><false part>?

<true part> ::= <APAL construct>*

<false part> ::= ELSE<newline><true part> | ELSE_IF<condition><choice> |
                 ELSE_IFN<condition><choice>


<APAL construct> ::= <module declaration> |
                     <module header> |
                     <module end> |
                     <global data identity> |
                     DEFINE<newline> |
                     END<newline> |
                     <data section> |
                     <data header> |
                     <data body> |
                     <data end> |
                     <code section> |
                     <code header> |
                     <code body> |
                     <code end> |
                     <mixed section> |
                     <mixed header> |
                     <VAR statement> |
                     <SET statement> |
                     <IF construct> |
                     <LIST statement> |
                     <NOTE statement> |
                     <STACK statement> |
                     <ERASE statement> |
                     <macro definition> |
                     <macro construct> |
                     <macro call>

<condition> See section 11.4.2
```

11.4.1.2 Semantics

When the assembler encounters an IF construct, assembly proceeds as follows:

- The assembly-time condition in the IF or IFN is evaluated (see section 11.4.2 for details)
- If the condition in an IF is true, or if the condition in an IFN is false any following APAL statements up to, but not including, the matching ELSE, ELSE_IF, ELSE_IFN, or FI are assembled. Assembly then continues with the statement following the matching FI.

  If the condition in an IF is false, or if the condition in an IFN is true any following APAL statements up to, but not including, the matching ELSE, ELSE_IF, ELSE_IFN, or FI are skipped. Assembly continues as shown in the remaining notes
- If the statement after the skipped ones is an ELSE, any following APAL statements up to, but not including, the matching FI are assembled
- If the statement after the skipped ones is an ELSE_IF or ELSE_IFN, this statement and those up to and including the matching FI are treated as an IF construct and are assembled by the process described above

The statements that are assembled as a result of an IF construct can themselves contain IF constructs and macro definitions, provided that the entire IF construct or macro definition is contained within the statement sequence.

An example of the use of the IF construct is:

IF *condition*$_1$
.
.
.
statement-sequence-$_1$
.
.
.
ELSE_IFN *condition*$_2$
.
.
.
statement-sequence-$_2$
.
.
.
ELSE
.
.
.
statement-sequence-$_3$
.
.
.
FI

which will assemble the statement sequences as follows:

condition$_1$	*condition$_2$*	*APAL statements assembled*
True	True	*statement-sequence-$_1$*
True	False	*statement-sequence-$_1$*
False	True	*statement-sequence-$_3$*
False	False	*statement-sequence-$_2$*

11.4.2 Assembly-time conditions

An assembly-time condition is a logical relationship between assembly-time values that evaluates to a logical truth value.

11.4.2.1 Syntax

<condition> ::= <arithmetic test><test-operand>,<test-operand><newline> |
                <monadic string test><string><newline> |
                <diadic string test><string>,<string><newline>

<arithmetic test> ::= GT | GE | EQ | LE | LT | NE

<test-operand> ::= <number> | <signed integer>

<monadic string test> ::= EX | ID | NU | VA | AV | VS

<diadic string test> ::= BE | SA

<number> ::= [<assembly-time expression>] | <unsigned integer>

<numval> ::= <number> | <hexadecimal value>

11.4.2.2 Semantics

The assembly-time conditions that occur in IF, IFN, ELSE_IF, or ELSE_IFN statements can be any of the following:

arithmetic-test operand$_1$, operand$_2$
monadic-string-test string$_1$
diadic-string-test string1, string$_2$

where:

operand$_1$ and *operand$_2$* are optionally signed integers or assembly-time expressions in [ ] yielding integer values. *operand$_1$* is delimited by a comma; *operand$_2$* is delimited by an '!' or a <newline>

arithmetic-test can have the following values and effects:

arithmetic-test	*Effect*
GT	True if *operand*$_1$ is greater than *operand*$_2$, otherwise false
GE	True if *operand*$_1$ is greater than or equal to *operand*$_2$ otherwise false
EQ	True if *operand*$_1$ equals *operand*$_2$, otherwise false
NE	False if *operand*$_1$ equals *operand*$_2$, otherwise true
LE	True if *operand*$_1$ is less than or equal to *operand*$_2$ otherwise false
LT	True if *operand*$_1$ is less than *operand*$_2$, otherwise false

string$_1$ of a monadic string test and *string*$_2$ of a diadic string test are delimited by ! or <newline>; *string*$_1$ of a diadic string test is delimited by a comma

monadic-string-test can have the following values and effects:

monadic-string-test	*Effect*
EX	True if *string*$_1$ is a valid assembly-time expression (note that *string*$_1$ does not include the surrounding [ ]), otherwise false
ID	True if *string*$_1$ is an identifier other than a macro name, macro parameter name, macro variable name, or assembly-time variable name, otherwise false. Only those identifiers in scope are considered
NU	True if *string*$_1$ is the null string, otherwise false
VA	True if *string*$_1$ is a value as defined in section 2.3 otherwise false
VS	True if *string*$_1$ is a value as defined in section 2.3 optionally followed by *size* as defined in section 4.2, otherwise false
AV	True if *string*$_1$ is the name of an assembly-time variable that is in scope, otherwise false

diadic-string-test can have the following values and effects:

diadic-string-test	*Effect*
BE	True if *string*$_1$ begins with the characters in *string*$_2$, otherwise false
SA	True if *string*$_1$ is the same as *string*$_2$, otherwise false

Examples of assembly-time conditions

BE ABCDEF, ABC is true.

EX 12*A+9 is true; note however, that EX [12*A+9] is false.

SA ABC, A_B_C is true, since the assembler removes underscore characters from strings.

VA 12.96E-9 is true, since the string is a valid real value.

VS 12.96E-9 is true.

VS 26(12) is true.

An example of the use of the IF construct

```
VAR A
.
.
.
IF GE [A], 63
SET A = 0

ELSE
SET A = [A + 1]
FI
```

Chapter 12

Macros

In common with most computer languages APAL provides a facility whereby you can associate a name with a sequence of APAL statements. Such a sequence, referred to as a *macro definition*, can be assembled at any subsequent point into an APAL program by writing the name associated with the macro definition; this is referred to as a *macro call*. The effect of a macro call is to replace the line on which the call appears with the statement sequence associated with the name in the macro definition. You can control the way in which statements from the macro definition are incorporated into the source program at assembly time by the use of *macro parameters* and *macro variables*.

This chapter describes the APAL macro facility and those assembly-time facilities that you can only use within macro definitions.

Simple examples of the use of a macro are given at the end of this chapter (in section 12.8) and in appendix C at the back of the manual.

12.1 Defining macros

A macro definition allows you to associate a name with a sequence of APAL statements.

12.1.1 Syntax

<macro definition> ::= <macro header><macro body><macro end>

<macro header> ::= MACRO<macro name><parameter template>?<newline>

<macro name> ::= <identifier>

<parameter template> ::= <formal parameter> | <formal parameter>?,<parameter template>

<formal parameter> ::= <macro parameter name><preset>?

<macro parameter name> ::= <identifier>

```
<preset> ::= =<assembly-time value>

<macro body> ::= <APAL construct>*

<macro construct> ::= <MEXIT statement> |
                      <MQUIT statement> |
                      <CYCLE statement> |
                      <MVAR statement> |
                      <MSET statement> |
                      <macro comment>

<macro comment> ::= !<comment><newline>

<macro end> ::= MEND<macro name>?<newline>
```

Note that <APAL construct> is defined in section 5 of appendix D.

12.1.2 Semantics

A macro definition associates an identifier with a sequence of APAL statements that is to be incorporated into the source program at a later point in the assembly process. A macro definition consists of:

- A *macro header*. The macro header marks the start of a macro definition, names it, and optionally specifies a formal parameter list – known as a parameter template
- The *macro body*, consisting of the sequence of APAL statements to be assembled into the program whenever the macro is called. A macro body can contain calls to other macros; it can also contain complete macro definitions
- A *macro end*, consisting of the line:

  MEND *macro-name*

  where *macro-name*, if specified, must be the same as the identifier in the corresponding macro header

Macro definitions can appear:

- Outside a module, in which case the macro can be called at any time during the assembly after the macro definition appears
- Anywhere within a module, in which case the macro can only be called from within that module, and after the macro definition has appeared

Any substitutions specified in the macro header are made when the macro is defined, but no substitutions take place in the macro body until the macro is called.

A nested macro definition is not recorded in the assembler symbol tables until the macro definition that contains it is called. The nested macro can only be called after the first such call of the outer macro. Each call of the outer macro causes a new definition of the nested macro to be processed, thereby rendering earlier definitions inaccessible. No substitutions occur within the body of a nested definition, until the nested macro is itself called.

There are no restrictions on the names of macros, although:

- If a macro name is the same as one of the APAL instruction mnemonics or APAL keywords listed in appendix A, any statement that begins with the macro name, after the macro definition, is recognised as a call of that macro until the macro name is either out of scope or erased (see section 12.1.3).

  The assembler will output a comment message for each macro definition in which such a name clash occurs

- If a macro name is the same as that of a previously defined macro that is still in scope, the assembler will output a comment to the effect that a name clash has occurred. The assembler will use the most recent definition of the macro until it is either out of scope or erased.

12.1.2.1 Parameter templates

The *parameter template* in a macro definition:

- Specifies identifiers for some or all of the formal parameters, so that the formal parameters can be referenced by name within the macro body
- Associates a parameter number with each formal parameter, so that the formal parameter can be referenced by number within the macro body
- Optionally supplies a default value for some or all of the formal parameters. If a call of the macro does not supply a value for a particular parameter, the parameter will be given this default value, if specified

A parameter template has the form:

$parameter_1$, $parameter_2$, ... $parameter_n$

where $parameter_i$ can be any of the following:

parameter-name
parameter-name = *assembly-time-value*
the null string

Formal parameters are separated by commas; the last parameter is terminated by the end of the statement.

In the first form of parameter specification the formal parameter is given a default value of the null string. In the second form the formal parameter is given a default value of a string or an assembly-time expression, as described in section 11.1. Both forms also specify the relative positions of the formal parameters.

The third form serves only to define the relative position of subsequent formal parameters.

For example:

```
MACRO MACEXAMPLE1 PAR1 = 0, PAR2, ,PAR3 = NAME,
```

defines a parameter template with five parameters, as follows:

Parameter name	*Parameter number*	*Default value*
PAR1	1	0
PAR2	2	Null string
–	3	Null string
PAR3	4	NAME
–	5	Null string

Note that you don't use the parameter template to define the number of actual parameters that can be passed to the macro; you use it to define those formal parameters that you want to reference by name within the macro body and to assign default values where required. If you have defined a parameter template then when you call the macro you can supply actual parameters by name, possibly in a different order from the one in the template (see section 12.3.2.1).

In any one macro definition, the names of macro parameters must all be different, and must also differ from any macro variables declared in the macro body (see section 12.5).

You can include the value of a macro parameter in the text of the macro body by a parameter substitution (see section 12.4).

Examples

-
  ```
  MACRO MACEXAMPLE2                   ! CONTINUATION LINES FOLLOW
  - PAR1 = [3*4],
  - PAR2 =                            ! NULL STRING
  - ,,                                ! FIRST COMMA TERMINATES PAR2
  - PAR3 =
  - STRING VALUE ON SEPARATE LINE
  ```

  This defines a macro with four parameters, as follows:

Parameter name	*Parameter number*	*Default value*
PAR1	1	12
PAR2	2	Null string
–	3	Null string
PAR3	4	STRING VALUE ON SEPARATE LINE

-
  ```
  MACRO MACEXAMPLE3                   ! INVALID CASE
  - PAR1 = AN INVALID STRING
  - ACROSS TWO LINES
  ```

12.1.3 The ERASE statement

The ERASE statement allows you to erase a particular macro definition from the assembler symbol tables.

12.1.3.1 Syntax

<ERASE statement> ::= ERASE<macro name><macro name>*<newline>

12.1.3.2 Semantics

An ERASE statement can appear anywhere in the APAL source, and has the form:

ERASE *macro-name*$_1$ *macro-name*$_2$ ... *macro-name*$_n$

where each *macro-name*$_i$ is the name of a previously defined macro. If that macro does not exist or is not in scope, the assembler flags an error.

The effect of ERASE is to make the most recent definition of each of the named macros permanently inaccessible. If the same macro is defined more than once ERASE allows you to make a previous version of the macro definition current again.

12.2 Calling macros

When the assembler encounters a macro call, assembly proceeds as if the macro body were substituted for the macro call into the source text at that point.

12.2.1 Syntax

<macro call> ::= <macro name><actual parameter list>?<newline>

<actual parameter list> ::= <actual parameter> | <actual parameter>, <actual parameter list>

<actual parameter> ::= <assembly-time value> |
<macro parameter name> = <assembly-time value>

12.2.2 Semantics

If the first item in an APAL statement is the name of a macro that is currently in scope, the statement is recognised as a macro call. The macro body associated with the name is assembled as if it had been written at that point instead of the macro call.

The macro name in the call is followed by an optional list of *actual parameters*, assembly-time values that are to be passed to the macro. These actual parameters need not correspond in number or order to the formal parameters defined in the parameter template of the macro.

If you nest macro calls (including calling the same macro recursively), the system creates a separate set of macro parameters for each level of nesting, and only the parameters associated with the current macro level are accessible.

You can supply actual parameters to a macro in two ways:

- By specifying them positionally, by supplying a list of assembly-time values in the macro call. The first such value becomes the value of the first parameter in the macro parameter template, the second value the value of the second parameter in the template, and so on
- By specifying them by name, by supplying a list of parameters each of the form:

  name = assembly-time-value

  You can only use this method of supplying actual parameters if a template was specified in the macro definition, and each instance of *name* matches an instance of *name* in the template, in which case each named parameter is given the specified value

If the macro definition includes a template, then if any parameters are not given values by either of the above methods in the macro call, and if values are specified in the template for those parameters, then the parameters take those template values by default.

You are recommended not to mix the above 'by name' and 'by position' methods of supplying macro parameter values, but if you do use both, it is best to give all the 'by position' values first, followed by the 'by name' values. If you *do* follow a named parameter and its value by one or more positional parameter values, then if the named parameter is n^{th} in the list in the formal parameter list in the template, then the positional parameter values that follow in the macro call will be assigned to the formal parameters numbered $(n + 1)$, $(n + 2)$, and so on.

Examples

- Given the macro definition:

```
MACRO EXAMPLE1 PAR1, PAR2=1, PAR3
.
.
MEND
```

  the macro call:

```
EXAMPLE1 PAR3 = 2*3
```

  creates the following parameters:

Parameter name	*Parameter number*	*Parameter value*
PAR1	1	Null string
PAR2	2	1
PAR3	3	2*3

- Given the macro definition:

```
MACRO EXAMPLE2 PAR1,, PAR2, PAR3
.
.
.
MEND
```

the macro call:

```
EXAMPLE2 1,2,3,4
```

creates the following parameters:

Parameter name	*Parameter number*	*Parameter value*
PAR1	1	1
-	2	2
PAR2	3	3
PAR3	4	4

- Given the macro definition:

```
MACRO EXAMPLE4 PAR1 = 3, , PAR2 = 4, PAR3 = 5
.
.
.
MEND
```

the macro call:

```
EXAMPLE4 PAR1 = 1, PAR2 = 0, 3
```

creates the following parameters:

Parameter name	*Parameter number*	*Parameter value*
PAR1	1	1
-	2	Null string
PAR2	3	0
PAR3	4	3

Note that all the defaults for the formal parameters are overwritten. In particular, PAR3 is assigned the value 3, which is given 'by position' after the 'by name' PAR2=0

12.2.3 System variables associated with macros

Two system variables 'MCOUNT and 'PCOUNT are associated with macros, and their values can be substituted using:

> % *system-varable-name*

in the same way as the other system variables discussed in section 11.3.2. Details of 'MCOUNT and 'PCOUNT are:

'MCOUNT	'MCOUNT is a variable containing a character string representing the number of macro calls processed within the module up to the time when 'MCOUNT was referenced. The number is zero initially, and is incremented at each macro call. 'MCOUNT can be referenced inside or outside a macro
'PCOUNT	The variable 'PCOUNT contains a character string representing the highest parameter number of all the actual parameters supplied in a macro call. If all parameters are supplied 'by position', then the number is simply the number of actual parameters in the call. Hence for a macro definition of: MACRO MIKE PAR1, PAR2, PAR3 . . MEND a macro call of MIKE 3, 4 would result in 'PCOUNT containing the string '2', and a call of MIKE PAR3=6,5,6 would result in 'PCOUNT containing the string '5'. You should only reference 'PCOUNT from within a macro; if you try to reference it outside a macro the assembler will flag an error

12.3 Leaving macros

You can exit from a macro by using any of the statements MEND, MEXIT, or MQUIT; MEXIT and MQUIT are usually associated with conditional assembly within the macro (see section 11.4 for details of conditional assembly).

12.3.1 The MEND statement

12.3.1.1 Syntax

<macro end> ::= MEND<macro name>?<newline>

12.3.1.2 Semantics

The MEND statement performs two functions:

- When the assembler encounters a macro definition, the macro body is taken as all those statements up to, but excluding, the corresponding MEND (that is, ignoring matching nested pairs of MACRO/MEND statements). If MEND specifies a name, it must be the same as the name specified in the corresponding MACRO statement
- When MEND is encountered following assembly of a macro body as a result of a macro call, assembly of that macro ceases; that is, assembly continues with the statement following the macro call. All macro variables and parameters defined in that macro then cease to exist

12.3.2 The MEXIT statement

12.3.2.1 Syntax

<MEXIT statement> ::= MEXIT<newline>

12.3.2.2 Semantics

Any number of MEXIT statements can appear within the macro body. When encountered during the assembly of a macro body as the result of a macro call MEXIT has the same effect as MEND. MEXIT has no effect when the macro definition is being processed.

12.3.3 The MQUIT statement

12.3.3.1 Syntax

<MQUIT statement> ::= MQUIT<newline>

12.3.3.2 Semantics

Any number of MQUIT statements can appear within the macro body. When MQUIT is encountered during assembly of a macro body as the result of a macro call, all current macro processing is abandoned; that is, assembly continues with the statement following the first macro call in the current macro calling sequence.

12.4 Parameter substitutions

You reference the value of a macro parameter within the macro body by a *parameter substitution*. The parameter values that are effective for a particular a macro call are constructed by the assembler, as explained in section 12.2.2.

Macro parameter substitutions can also be used in substring substitutions (see section 11.3.9 for details) and length substitutions (see section 11.3.10 for details). The method of using these substitutions is similar to the corresponding substitutions applied to assembly-time variables, with the addition for macro parameters of having the option of specifying parameters either by name or position, as discussed below. The 'OCCUR option of parameter substitution cannot be combined with substring or length substitutions.

12.4.1 Syntax

```
<parameter substitution> ::= %<macro parameter name> |
                             %^<macro parameter name> |
                             %<number> |
                             % ^<number> |
                             %'OCCUR[<number>,<string>]
```

12.4.2 Semantics

A parameter substitution performs a similar function to a variable substitution (see section 11.3.2), except that it can only appear within a macro.

A macro parameter can be referenced either *positionally* or by *name*.

You can reference any macro parameter positionally; that is, by specifying the relative position of the parameter within the formal parameter list.

A positional parameter substitution can have either of the following forms:

%*number*
%^*number*

where *number* is an unsigned integer, or an assembly-time expression in [ ] yielding a positive integer value. The first form is replaced by the string value corresponding to the n^{th} formal parameter, where n is the value of *number*. The second form has a similar effect, except that the string value of the parameter is inserted into the text in its unprotected form (see section 11.1.2.1). The value of *number* must not exceed the greater of the number of formal parameters in the macro definition and the value of 'PCOUNT (see section 12.2.3) for the current call of the macro. The first parameter is %1 (or %^1).

You can reference a macro parameter by name, provided that you have already associated an identifier with it in the parameter template. A name parameter substitution can have either of the following forms:

%*parameter-name*
%^*parameter-name*

The first form is replaced by the string represented by the named parameter. The second form is replaced by the unprotected form of the string value of the named formal parameter.

Example

Suppose you have as part of a macro definition:

```
    MACRO MARY PAR1,PAR2,PAR3
    .
    .
    QS %PAR1
    .
    .
    MEND
```

Later in the source code you might have a statement:

```
    MARY  50
```

The effect of the macro call would be that at assembly time the value 50 would be substituted for PAR1, thus creating the instruction QS 50. When at run time the QS instruction was reached, array store plane 50 would be loaded into the Q plane – see appendix F for details of QS.

A parameter substitution can also have the form:

%'OCCUR [*number, string*]

which is replaced by the remainder of the value of the n^{th} actual parameter beginning with *string*, where *n* is the value of *number*. If no such parameter can be found a null string is substituted. Parameters are searched in ascending order of parameter number.

For example, given the macro call:

```
    MAC1 123, 134, 216, 1357, 21, 9, 13
```

the substitution:

```
    %'OCCUR[3,1]
```

yields the string 357; that is, the remainder of the third parameter beginning with 1.

If either of *string* or the values of any of the parameters is a protected string, the algorithm described for substring substitutions (see section 11.3.9) is implemented. For example, given a macro call:

```
    MAC2 AB^C, A^BD, A^^BC, CD, ABE
```

the substitution:

```
    %'OCCUR [3, A^B]
```

yields the string E. Note that the fourth parameter ABE is the third parameter beginning with A^B or AB.

12.5 Macro variables

A macro variable is similar to an assembly-time variable except that it has only local scope within the macro that defined it. In the macro context there are the MVAR and MSET statements, corresponding to the VAR and SET assembly-time statements, which process assembly-time variables.

Macro variables can be used in the same way as assembly-time variables, and with the same syntax, in variable substitutions (see section 11.3.2 for further details), substring substitutions (see section 11.3.9) and length substitutions (see section 11.3.10).

12.5.1 The MVAR statement

The MVAR statement declares and optionally gives an initial value to a macro variable.

12.5.1.1 Syntax

<MVAR statement> ::= MVAR<macro variable name><preset>?<newline>

<macro variable name> ::= <identifier>

<preset> ::= =<assembly-time-value>

12.5.1.2 Semantics

MVAR can have either of the following forms:

MVAR *name*
MVAR *name = assembly-time-value*

MVAR specifies an identifier by which you can reference the macro variable from a subsequent point in the macro body. MVAR must precede any reference to the named macro variable. The identifier has local scope within the macro body and must be distinct from the name of any other macro variables or parameters in the macro definition. If the name is the same as that of an assembly-time variable currently in scope, the value of the assembly-time variable will not be accessible until the macro variable is out of scope; however, the assembly-time variable can be given a new value using the SET statement.

For example, if the following statement appears in a macro body:

```
SET A = [A + 1]
```

and you have declared a macro variable called A previously in that macro body, it is not clear whether you are referring to the assembly-time variable or the macro variable, so you are recommended to avoid the situation. In the case above the assembler will assume that you mean to refer to the assembly-time variable on the left of the = and to the macro variable on the right of the =.

A macro variable name is only in scope when control is within the macro in which the name is defined; the name is NOT in scope when control is within any macro called or defined by the name-defining macro. That is, if a macro variable name is declared in macro A, and macro A calls or defines macro B, then the name is out of scope when control is within macro B.

The values that you can assign to a macro variable are strings, including the null string, or assembly-time expressions enclosed in [ ] which yield integer values. You can reference the values of macro variables using variable substitutions (see section 11.3.2).

12.5.2 The MSET statement

The MSET statement assigns a new value to a previously-defined macro variable or macro parameter.

12.5.2.1 Syntax

```
<MSET statement> ::= MSET<macro variable name><preset><newline> |
                     MSET<macro parameter name><preset><newline> |
                     MSET<number><preset><newline>
```

12.5.2.2 Semantics

Once you have used MVAR to declare a macro variable, the variable's value can be changed by an MSET of the following form:

MSET *name = assembly-time-value*

You can reference any macro parameter positionally, that is, by using its relative position on the macro parameter stack; in this case the MSET has the form:

MSET *number = assembly-time-value*

For example:

```
MSET 4 = 126     ! FOURTH PARAMETER ON STACK IS THE STRING 126
```

If an identifier is associated with a parameter in the parameter template, you can also reference it by name, as follows:

```
MSET PAR1 = NEW VALUE       ! PARAMETER PAR1 IS GIVEN A STRING
                            ! VALUE OF 'NEW VALUE'.
```

Note that when you reference a macro parameter positionally, *number* must not exceed the greater of the number of formal parameters in the macro definition and the value of 'PCOUNT for the current call of the macro, otherwise an error is flagged by the assembler.

Example

The example on the next page shows a different method of using MSET to provide default parameter values.

```
MACRO A P1 = DEFAULT
.
.
.
MEND
```

can be replaced by:

```
MACRO A P1
IF NU%P1
  MSET P1 = DEFAULT
FI
.
.
.
MEND
```

12.6 The CYCLE construct

The cycle construct causes the repeated assembly of some part of a macro body.

12.6.1 Syntax

<CYCLE construct> ::= <cycle header><cycle body><cycle end>

<cycle header> ::= CYCLE<number>?<newline>

<cycle body> ::= <APAL construct>*<conditional part>?

<conditional part> ::= <cycle test> <APAL construct>*

<cycle test> ::= WHILE<condition><newline> | UNTIL<condition><newline>

<cycle end> ::= REPEAT<newline>

12.6.2 Semantics

You might need to assemble repeatedly a sequence of statements within a macro body, subject to a certain assembly-time condition. Rather than include each repetition of the sequence within the macro body, you can use CYCLE to specify the sequence just once.

The CYCLE construct, which can only appear within a macro body, has the general form:

CYCLE *number*
statement-sequence-1
WHILE (or UNTIL) *condition*

statement-sequence-2
REPEAT

where:

number, which is optional, is an unsigned integer, or an assembly-time expression within [ ], yielding a non-negative integer value

statement-sequence-1 and *statement-sequence-2*, which are optional, are sequences of APAL statements

WHILE (or UNTIL) *condition* is optional

condition is as defined in section 11.3.2.

When the assembler encounters CYCLE, assembly proceeds as follows:

- *number*, if specified, is evaluated. *number* determines the maximum number of times that the APAL statements in *statement-sequence-1* and *statement-sequence-2* are to be assembled.

  If *number* is negative or does not evaluate to an integer, an error is flagged and a value of 1 is assumed. If *number* is zero, assembly continues with the statement following REPEAT. If *number* is omitted, the number of repetitions of the assembly is determined by WHILE (or UNTIL) *condition*. *number* must be specified if WHILE (or UNTIL) *condition* is omitted

- If *number* is not zero, the APAL statements, if any, in *statement-sequence-1* are assembled. These statements will be assembled in every cycle

- If WHILE (or UNTIL) *condition* is present, *condition* is evaluated (see section 11.2.2). According to the result of the evaluation, and whether WHILE or UNTIL is specified, assembly continues as follows:

Test	*Condition*	*Effect*
WHILE	True	*statement-sequence-2* is assembled
WHILE	False	*statement-sequence-2* is ignored, and assembly continues with the APAL statement following REPEAT (the cycle is terminated)
UNTIL	True	*statement-sequence-2* is ignored, and assembly continues with the APAL statement following REPEAT (the cycle is terminated)
UNTIL	False	*statement-sequence-2* is assembled

  WHILE (or UNTIL) *condition* must be specified if *number* is omitted

- Provided the cycle has not been terminated by the WHILE (or UNTIL) *condition*, and the cycle has been through fewer than *number* cycles, the process repeats with the assembly of *statement-sequence-1*. If the maximum value of cycles has been reached, assembly continues with the APAL statement following REPEAT

Note that unless the value of *number* is zero, *statement-sequence-1* is assembled at least once. If the loop terminates because the WHILE condition is not satisfied (or the UNTIL condition *is* satisfied), then *statement-sequence-1* is assembled once more than *statement-sequence-2*.

statement-sequence-1 or *statement-sequence-2* can contain other CYCLE or IF constructs (see section 11.3), provided that the entire CYCLE or IF is completely contained within one of the statement sequences. The sequences can also contain macro definitions, again provided that the definitions are completely contained within a statement sequence.

Example

The following example shows how CYCLE can be used within a macro definiton to set the values of all parameters passed to it to null strings:

```
MACRO NULLSET
MVAR COUNT = 1
CYCLE                              ! NO NUMBER, SO 'WHILE' TEST ENDS CYCLE
WHILE LE %COUNT, %'PCOUNT
  MSET %COUNT =                    ! SET PARAMETER TO NULL STRING
  MSET COUNT = [COUNT + 1]
REPEAT
MEND NULLSET
```

12.7 Macro comments

When a macro is called, any comments that form part of that macro definition are treated in the same way as the other text of the macro, subject to the LIST option. However, *macro comments* are suppressed when the macro is called, and are only printed when the macro definition is listed – and the appropriate LIST option is set.

12.7.1 Syntax

<macro comment> ::= !<comment><new line>

<comment> ::= !<comment character>*

<comment character> ::= <basic character> | ^<special character>

12.7.2 Semantics

A macro comment is only recognised as such if it is the first and only item on a line, and begins with the two characters '!!'. The comment will only be listed if the macro definition itself is listed; it will not be incorporated into the source when the macro is called.

If a macro comment appears after or within an APAL statement, it is treated as a normal comment.

For example:

```
MACRO A
                !! EXAMPLE OF A MACRO COMMENT
RR M1 M2                                  !! THIS IS A NORMAL COMMENT
.
.
.
MEND
```

12.8 Example of a simple macro

The definition of a macro to add two arrays of matrices might be:

```
MACRO ADD_MAT MAT1, MAT2, N_BITS
                  !! ADDS TWO MATRIX ARRAYS, LEAVING THE RESULT
                  !! IN THE FIRST ARRAY; MCU REGISTERS MAT1 AND MAT2
                  !! SPECIFY THE START ADDRESSES OF THE TWO ARRAYS.
                  !! EACH ARRAY CONSISTS OF N_BIT PLANES
MVAR  LAYER  = %N_BITS
    CF
    AT
    CYCLE
    MSET  LAYER  =  [LAYER - 1]
    WHILE  GE  %LAYER, 0
        QS          %LAYER (%MAT2)
        SICPCQS     %LAYER (%MAT1)

    REPEAT

MEND
```

If in your APAL code you wanted to add two matrix arrays each 8 planes deep and whose start addresses in array store were held in register M2 and M4, say, then you could issue the macro call:

```
ADD_MAT M2, M4, 8
```

and the resultant array would be put in the 8 planes of store pointed to by M2.

Appendix A

APAL Keywords

A number of valid APAL identifiers have a special significance in APAL; such identifiers are called *APAL keywords.* Although you can use an APAL keyword as an identifier, such usage is not recommended in the interests of program clarity. In particular, you should be aware of the consequences of declaring a macro whose name is the same as an instruction mnemonic or assembly-time statement (see chapter 12).

APAL keywords fall into three classes: instruction mnemonics, system variables and functions, and other keywords.

A.1 APAL instruction mnemonics

AB	AND	CPCQR	CQPCA	CQPQRNO
ABN	ANDH	CPCQRN	CQPCAN	CQPQRO
ADD	ANDHN	CPCQRNO	CQPCQ	CQPQS
ADDC	AQ	CPCQRO	CQPCQA	CQPQSN
ADDH	AQN	CPCQS	CQPCQAN	CQ_QQN
ADDHC	AQ_QQ	CPCQSN	CQPCQR	CQVCQ
AEBS	AR	CPCQT	CQPCQRN	CVCQ
AEBSN	ARN	CPCR	CQPCQRNO	
AF	ARNO	CPCRN	CQPCQRO	DECR
AMB	ARO	CPCRNO	CQPCQS	DO
AMBN	AS	CPCRO	CQPCQSN	
AMEBS	AS_CF	CPCS	CQPCQT	EQV
AMEBSN	ASN	CPCSN	CQPCR	EQVH
AMQ	ASN_CF	CPQA	CQPCRN	EXIT
AMQN	AT	CPQAN	CQPCRNO	
AMQ_QQ		CPQR	CQPCRO	INCR
AMR	CF	CPQRN	CQPCS	
AMRN	CPCA	CPQRNO	CQPCSN	J
AMRNO	CPCAN	CPQRO	CQPQA	JE
AMRO	CPCQ	CPQS	CQPQAN	JESL
AMS	CPCQA	CPQSN	CQPQR	JSL
AMSN	CPCQAN	CQ	CQPQRN	

LOOP	QCN	QQN	RR	SKIP
	QEBS	QR	RRN	SQ
MPY32	QEBSN	QRN	RS	SQ_AQ
MPY32V	QF	QRNO	RSO	SQ_CQ
MPY64	QF_AF	QRO	RT	SQ_QC
MPYU32	QF_CF	QS	RW	SQ_QCN
MPYU32V	QPCA	QS_AS	RWO	SQ_QF
MPYU64	QPCAN	QS_CF	RX	SQ_QT
	QPCQ	QSN	RXO	SR
NAND	QPCQA	QSN_ASN		SRN
NANDH	QPCQAN	QSN_CF	SAN	STOP
NANDHN	QPCQR	QT	SF	SUB
NEQ	QPCQRN	QT_AT	SHL	SUBC
NEQH	QPCQRNO	QT_CF	SHLC	SUBH
NOR	QPCQRO	QVCQ	SHR	SUBHC
NORH	QPCQS		SHRA	
NORHN	QPCQSN	RAC	SHRC	WF
NULL	QPCQT	RACE	SIC	WR
	QPCR	RALITR	SICPCQS	WRN
OR	QPCRN	RALITW	SICPCS	
ORH	QPCRNO	RANO	SICPQS	XAN
ORHN	QPCRO	RAPL	SICQPCQS	XF
	QPCS	RAR	SICQPCS	XIC
PAUSE	QPCSN	RASC	SICQPQS	XIF
	QPQA	RAW	SIF	XIPCQ
QA	QPQAN	RAWD	SIPCQ	XIQ
QA_CF	QPQR	RAX	SIPCQS	XQ
QAN	QPQRN	RDGC	SIPCS	XR
QAN_CF	QPQRNO	RF	SIPQS	XRN
QB	QPQRO	RH	SIQ	
QBN	QPQS	RHN	SIQPCQS	
QC	QPQSN	RLIT	SIQPCS	
QC_CF	QQ	RQO	SIQPQS	

A.2 System variables and functions

There are 35 keywords used by the system for various functions and variables; all of them start with a ’ (pronounced ‘blip’); they are listed below. You are strongly recommended not to use any identifier names starting with a ’.

’AND	’CO	’COLUMN	’DA	’DATE
’DI	’DIM	’LE	’LENGTH	’LO
’LOGDIM	’MC	’MCOUNT	’NEQ	’OC
’OCCUR	’OR	’PC	’PCOUNT	’PL
’PLANE	’RO	’ROTATE	’ROW	’SCALE
’SE	’SECTION	’SU	’SUBSTRING	’TI
’TIME	’TR	’TRANSFER	’WO	’WORD

A.3 Other keywords

A	EQ	M10	MIXED	SA
ALIGN	ERASE	M10N	MODULE	SET
ALL	ERROR	M11	MQUIT	SHORT
ANY	EX	M11N	MSET	SIZE
AV		M12	MVAR	SOURCE
	F	M12N		STACK
BE	FI	M13	N	
BIT	FROM	M13N	NE	T
BITS	FROMBIT	M1N	NONE	TERMINAL
	FULL	M2	NOTE	TIMES
C		M2N	NU	TRACE
CARRY	GE	M3		TYPE
CHAR	GT	M3N	P	
CODE		M4	PC	UNTIL
COL	HEX	M4N	PER	UPTO
COLS	HOST	M5	PLANE	
COMMENT		M5N	PLANEALIGN	V
COMMON	I	M6	PLANES	VA
CP	ID	M6N		VAR
CYCLE	IF	M7	R0	VERTICAL
DAP	IFN	M7N	R1	VS
DATA	INT	M8	R2	
DEFINE		M8N	R3	W
	LE	M9	REAL	WARNING
E	LEVEL	M9N	REPEAT	WHILE
ELSE	LIST	MACRO	ROW	WORD
ELSE_IF	LT	MCUR	ROWALIGN	WORDPACK
ELSE_IFN		ME	ROWPACK	WORDS
END	M0	MEND	ROWS	WRITE
ENDMODULE	M0N	MER		
ENTRY	M1	MEXIT	S	

Appendix B

Character set

This appendix lists those characters that may appear in APAL source text, and gives the internal hexadecimal representation of each character. The following table shows the ASCII character set, used by the APAL assembler and the DAP run-time software.

Character	*Hexadecimal*	*Character*	*Hexadecimal*
(space)	20	:	3A
!	21	;	3B
" (double quotes)	22	<	3C
# (or £)	23	=	3D
$	24	>	3E
%	25	?	3F
&	26	@	40
' ('blip' or single quotes)	27	A	41
(	28	B	42
)	29	C	43
*	2A	D	44
+	2B	E	45
, (comma)	2C	F	46
- (hyphen)	2D	G	47
.	2E	H	48
/	2F	I	49
0	30	J	4A
1	31	K	4B
2	32	L	4C
3	33	M	4D
4	34	N	4E
5	35	O	4F
6	36	P	50
7	37	Q	51
8	38	R	52
9	39	S	53

Character	*Hexadecimal*
T	54
U	55
V	56
W	57
X	58
Y	59
Z	5A
[	5B
\	5C
]	5D
^ (circumflex)	5E
_ (underline)	5F
` (grave)	60
a	61
b	62
c	63
d	64
e	65
f	66
g	67
h	68
i	69
j	6A
k	6B
l	6C
m	6D
n	6E
o	6F
p	70
q	71
r	72
s	73
t	74
u	75
v	76
w	77
x	78
y	79
z	7A
{	7B
| (vertical bar)	7C
}	7D
~ (tilde)	7E

Appendix C

Examples of APAL code

This appendix starts with three examples of fragments of APAL code, each designed to carry out a fairly simple task. The appendix ends with a complete DAP program that makes extensive use of APAL facilities; it also includes the associated host program.

C.1 Code fragment for matrix addition

First, a fragment of code fragment that will carry out element-by-element addition of two 16-bit integer matrices. Each of the two operands for the addition is held in 16 array store planes, pointed to by MCU registers M2 and M3; the result of the addition is to be put in the 16 array store planes pointed to by register M1. Only the plane part of the addresses in M1, M2 and M3 is relevant, and by convention the lowest numbered of each of the 16 planes contains the most significant bit of each of the integers. The contents of the PE registers are initially undefined. The code is as follows:-

```
!
   CF                     ! Clear the Carry register
!
   DO   16  TIMES
     QS       15(M2-)     ! Load a bit from the first operand into the Q plane
     CQPCQS   15(M3-)     ! Add a bit from the second operand into the Q plane
     SQ       15(M1-)     ! Store the result bit
   LOOP
!
```

The 'DO . . . LOOP' construct invokes the hardware-controlled loop over the instructions included within it. The loop terminator 'LOOP' is a pseudo-instruction that generates no code, but tells the assembler how many instructions are included within the loop. DO is an instruction implemented directly in hardware, and the assembler encodes within it the number of times that the loop is to be executed (information supplied by you; 16 in this case) and the number of instructions within the loop (information it works out; three in this case).

Instruction 'QS' loads the Q register in each PE with a bit from an array store plane (that is, copies a store plane into the Q plane). The address of the store plane is held in M2 in this case, and the offset of 15 planes specifies that the (by convention) least significant bit of the number is accessed. Mnemonic CQPCQS is the general form of the add (or Plus) operation, where the operands (C, Q and S in this case) are to the right of the P and the results (carry to C, sum to Q) to the left.

Automatic stepping of memory addresses is specified by the '-' as a qualifier to the address modifier, meaning that the effective plane address is decremented each time round the loop. Thus on successive passes of the loop the plane addresses are 15, 14, 13, . . . , 1, 0, all relative to the plane address held in the MCU register.

At the end of the code extract above, the carry out from the most significant bit of the addition is in the C register, and normally further code would perform an overflow check, the details depending on whether the operands are regarded as signed or unsigned values.

C.2 Code fragment to extract and broadcast rows from array store

This example extracts a row from a matrix of 32-bit values and replicates the row's values in every row of the resultant 32 array store planes. Register M2 holds the address of the operand, and both the plane part of the address and the row-within-plane part are used. Register M1 holds the starting address in array store where the result is to be put. The code is:

```
!
  DO  32  TIMES
    RX  ME  0.0(M2+A)          ! Select a row to be broadcast
    SR  ME  0(M1+)             ! Broadcast the row to an array store plane
  LOOP
!
```

Instruction RX extracts a row from a plane in array store, and writes the result into the edge-size register, ME. Instruction SR broadcasts the register to each row in the notional R-plane, and copies the R-plane to an array store plane, in this case pointed to by register M1. The most significant bits of the 32-bit values have been dealt with first in this example, though the code could equally have been written to start at the least significant end of the values. The address stepping for the SR instruction is similar to that in the example in section C.1.

The RX instruction uses the row part of the address within M2 as well as the plane part. In general both a plane and a row offset can be specified, but here they are both 0, so the offset is written as '0.0' (which could have been written simply as '0', but by using '0.0' you underline that a row address is being used). When you access a row there are two ways of stepping the address: successive rows in the same plane, or successive planes but the same row. In this case the latter is required, so address stepping is specified as '+A'. If, for example, M2 contains the address 506.5 (plane 506 row 5), then the effective addresses on successive passes of the loop would be 506.5, 507.5, 508.5, and so on.

C.3 A fragment using macro facilities

The piece of code in C.2 has been rewritten as a macro having as parameters the precision of the values in the row to be replicated, and the MCU registers holding source and destination addresses:

```
!
MACRO  ROWREPLICATE   FROM = M2 , TO = M1 , LEN = 32
       DO  %LEN  TIMES
         RX   ME  0.0(%FROM+A)
         SR   ME  0(%TO+)
       LOOP
MEND
!
```

The first line defines the name of the macro, ROWREPLICATE in this case, and defines names and default values for its three parameters. Thus when the macro is invoked, the actual value of the third formal parameter is subtituted for %LEN in the body of the macro, and so on. Since all the parameters in the macro have been given default values, any of the parameters can be omitted from a macro call, in which case they would be given their default values. Thus a call of ROWREPLICATE with no parameters given in the call would generate exactly the code given in C.2 above.

Code to replicate 28-bit values, where the address of the operand is in M3 and the destination in M1, can be generated by the macro call:

```
ROWREPLICATE  FROM = M3 , TO = M1 , LEN = 28
```

or more simply, since TO is given a default value of M1 in the parameter template in the macro definition:

```
ROWREPLICATE  FROM = M3 , LEN = 28
```

C.4 Complete program: Conway's Game of Life

This complete program includes both the DAP and Host parts. The program calculates successive generations of a colony of cells on a cyclic grid, using the rules of Conway's Game of Life. At any stage in the Game the next generation at a grid point depends on the number of its eight nearest neighbours which hold live cells.

The rules specify that at any grid point, if there are three 'live' nearest-neighbours – whether or not the point itself is live – or if the point *is* live and two of its nearest-neighbours are live, then the point is live in the next generation of the game. If these conditions are not met, then the point is 'dead' in the next generation.

The state of each grid point is held on the DAP in the 1-bit matrix LASTGEN as either **true**, that is, live in Game of Life terms, or **false** or dead. The central part of the program is the DAP entry subroutine UPDATE. Each time UPDATE is called it advances the pattern held in LASTGEN by one timestep. This updating involves at every grid point:

- Summation of the number of **true** (or live) values in the eight nearest-neighbour points, using cyclic wraparound at the edges
- Updating the value at the point according to the rules of Conway's Game of Life

Subroutine CONVERT changes the initial data pattern from a format convenient to the host into the 1-bit matrix expected by UPDATE. Subroutine RECONVERT performs the inverse transform, and is called after each timestep to allow the host to print out the current pattern. Addresses of operand and result are passed as parameters to RECONVERT in order to illustrate the use of parameter passing; in practice addresses could in this case be dealt with in the same way as in CONVERT.

The host program LIFE inputs the initial data to the DAP, then repeatedly calls UPDATE, each time receiving back from the DAP the new pattern and then printing it out. Further details of the interface subroutines DAPSEN, DAPREC and DAPENT are given in *DAP Series: Program Development.*

The program listed below 'plays' the Game for a 32 x 32 grid of points, and was written to run on a DAP 500. The subroutine UPDATE would play the Game for a 64 x 64 grid if the code were re-compiled to run on a DAP 600, but the DAP conversion routines and the host program would need to be changed to allow for the larger number of data items.

Additional features could easily be added to the code. For example, the DAP program could keep a record of the last few generations to see if a repetitive loop had been entered. It would also improve performance if the DAP program allowed several generations to elapse before returning to the host program to print the current state.

Host Program

```
      PROGRAM LIFE
C
C  Conway's Life Game
C
      COMMON/MATDAT/START
      COMMON/MATRES/RESULT
C
      INTEGER START(32,32), RESULT(32,32)
      INTEGER DAPCON
C
C  Read start position from file. The state of each of the 32x32 grid of cells
C  is stored as integer value 0 or 1, representing dead and live cells
C  respectively.
C
      READ (5,99) ((START(I,J), J=1,32), I=1,32)
   99 FORMAT (32I1)
C
```

```
C
C  Print out generation 0
C
         CALL SHOW (START,0)
C
C  Connect to DAP Program in object file 'LIFE_DOF'
C
         IF (DAPCON ('LIFE_DOF') .NE. 0) STOP
C
C  Send START to 1024 words of data section MATDAT in the DAP program
C
         CALL DAPSEN ('MATDAT', START, 1024)
C
C  Call the DAP program at code section CONVERT to prepare the data
C
         CALL DAPENT ('CONVERT')
C
C  The main loop calls the DAP program at code section UPDATE to calculate the
C  next generation of cells, receives it in RESULT, and prints it.
C
C
         DO 10 NUM = 1,1000
         CALL DAPENT ('UPDATE')
         CALL DAPREC ('MATRES',RESULT,1024)
         CALL SHOW   (RESULT,NUM)
      10 CONTINUE
C
         STOP
         END

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

         SUBROUTINE SHOW(MATRIX,GEN)
C
C  Display the pattern in MATRIX
C
         INTEGER MATRIX(32,32), GEN
         CHARACTER MAP(32,32)
C
         DO 10 J=1,32
         DO 10 I=1,32
         MAP(I,J) = '-'
         IF (MATRIX(I,J) .EQ. 1) MAP(I,J) = 'X'
      10 CONTINUE
C
```

```
C
      WRITE(6,100) GEN
  100 FORMAT(///28X,'GENERATION ',1I6///)
      WRITE(6,200) ((MAP(I,J),J=1,32),I=1,32)
  200 FORMAT(20X,32A1)
C
      RETURN
      END
```

DAP Program

```
MODULE DAP_PROGRAM
#include usrmacs.da
!
DATA MATDAT HOST COMMON WRITE
   32*PLANE
END
!
DATA MATRES HOST COMMON WRITE
   32*(32*0)
END
!
DATA LASTGEN WRITE
   PLANE
END

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

CODE CONVERT HOST
!
!  The data in MATDAT has been sent from the host program. It consists of 1024
!  rows, bit 31 of each row containing either 0 or 1.
!
!  The data in MATDAT is now to be compressed into a single plane LASTGEN, by
!  reading the column 31s of the successive planes of MATDAT, and copying
!  those columns to successive columns of LASTGEN.
!
'PROLOGUE
   RASC    M1  MATDAT                ! Put address of MATDAT into M1.
   RASC    M2  LASTGEN               ! Put address of LASTGEN into M2.
   QT_AT                             ! Set both plane Q and activity plane A
                                     ! to all TRUE.
!
```

```
!
DO 32 TIMES
   RXO     ME  0.31 (M1 + A)      !  Load a column of MATDAT into ME
   CPQRO   ME                     !  Copy ME into columns of C where Q TRUE
   SIC         0 (M2)             !  Write to LASTGEN under activity control
   AQ_QQ   E P 1                  !  SHIFT A, to protect previous columns
LOOP

'EPILOGUE                         !  Return to host program
END

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

CODE UPDATE HOST
!
!  This subroutine takes the pattern of cells in LASTGEN, and calculates the
!  next generation. To do this it needs the sum of all 8 neighbours at every
!  grid point.
                                  !  Local data identities to workspace area:
   SELFNS      =   0              !  2 planes for sum of self and North and
                                  !  South neighbours
   NEIGHBOURS  =   2              !  4 planes for sum of all neighbours.
!
'PROLOGUE
   RASC    M2  LASTGEN            !  Put address of LASTGEN into M1.
   AT                             !  Set activity plane A to all TRUE.
!
!  First calculate the sum of the North and South neighbours, and store the
!  answer (max 2 bits) in NEIGHBOURS.
!
   QS          0 (M2)             !  Put LASTGEN into Q plane
   QQ      N C 1                  !  Shift in South neighbour ...
   SQ          NEIGHBOURS+3       !  ... and store
   QQ      S C 2                  !  Shift in North neighbour ...
   SICPQS      NEIGHBOURS+3       !  ... and add
   SIC         NEIGHBOURS+2
!
!  Now calculate sum of North and South neighbours and self, and store answer
!  (max 2 bits) in SELFNS.
!
   QS          0 (M2)
   CQPQS       NEIGHBOURS+3       !  Add self ...
   SQ          SELFNS+1           !  ... and result to SELFNS
   QPCS        NEIGHBOURS+2
   SQ          SELFNS
!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
! Define a macro which adds SELFNS shifted 1 place to the least significant 2
! bits of NEIGHBOURS.
!
MACRO ADDIT DIRN
   QS          SELFNS+1
   QQ      %DIRN C 1           ! Direction of shift is a parameter
   SICPQS      NEIGHBOURS+3
   QS          SELFNS
   QQ      %DIRN C 1
   SICPCQS     NEIGHBOURS+2
MEND

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! Add the value in SELFNS to the west. This is the sum of the original west
! north-west and south-west neighbours (max 3 bits)
!
   ADDIT   E
   SIC         NEIGHBOURS+1
!
! Conceptually, the result in the C plane is written to NEIGHBOURS;
! in pratice it is left in C as it is needed again immediately and
! not needed later, so saving an instruction
!
   ADDIT   W
   SICPCS      NEIGHBOURS+1
!
! The rules are that the new generation of each cell is live (true) where:
! (NEIGHBOURS.EQ.3) .OR. (LASTGEN.AND.NEIGHBOURS.EQ.2),
! and dead (false) otherwise
!
! First calculate where NEIGHBOURS is 2 or 3.
! that is, test the bits in 3 of the 4 planes of NEIGHBOURS against
! the binary value 001x (x = don't care)
!
   QCN                         ! Q is .TRUE. if C-plane is .FALSE.
                               ! that is, if MSB of NEIGHBOURS is .FALSE.
   CPQSN       NEIGHBOURS+1
   CPCS        NEIGHBOURS+2
   QC                          ! hence Q is .TRUE. where NEIGHBOURS
                               ! is 2 or 3
!
```

```
!  Calculate (LASTGEN.AND.NEIGHBOURS.EQ.2)
!
   CPCSN       NEIGHBOURS+3        !  C is .TRUE. where NEIGHBOURS is 2
   CPCS        0 (M2)              !  C is (LASTGEN.AND.NEIGHBOURS.EQ.2)
!
!  Calculate (NEIGHBOURS.EQ.3)
!
   AS          NEIGHBOURS+3
   AMQ                             !  A is .TRUE. where NEIGHBOURS is 3
!
!  Finally calculate A.OR.C and save the answer in LASTGEN
!
   QPCA                            !  Create logical OR of C and A planes.
   SQ          0 (M2)              !  Save new generation in LASTGEN.
!
!  Now call a subroutine to expand the new value of LASTGEN into 1024 rows in
!  the data section MATRES.
!
   RAX     M5  1 (M7)              !  New name space
   WR      M2  'PARBASE..1 (M5)    !  Parameter 1 is address of LASTGEN
   RASC    M3  MATRES
   WR      M3  'PARBASE..2 (M5)    !  Parameter 2 is address of MATRES
   'CALLNAME   RECONVERT
   'EPILOGUE                       !  Return to the host program
   END

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

CODE RECONVERT
!
!  This subroutine takes 2 parameters - a plane and a set of 32 planes.
!  Successive columns of the single plane are to be written to the last column
!  of successive planes in the set.
!
!
'PROLOGUE 2
   RLIT    ME  1
   AR      ME                      !  Activity plane is true in column 31
DO 32 TIMES
   RXO     ME  0.31 (M2-)          !  Read a column in plane
   QRO     ME
   SIQ         31 (M3-)            !  Write to column 31 of one of set
LOOP
'EPILOGUE                          !  Return to calling routine
END
!
ENDMODULE
```

Appendix D

APAL syntax

This appendix is a summary of the APAL syntax in the remainder of this publication.

The following symbols are used in the syntax definition:

Symbol	*Meaning*
< >	Enclose the name of a syntactic construct (for example, <character>)
:: =	Is equivalent to the following syntactic construct(s)
|	Separates alternative forms of a syntactic construct
*	The immediately preceding syntactic construct may appear any number of times, or not at all
?	The immediately preceding syntactic construct is optional and may be omitted

The construct <space> represents a space character.
The construct <question> represents a question mark character.
The construct <star> represents an asterisk.
The construct <vertical bar> represents a vertical bar character.
The construct <new line> implies the end of a source statement; that is, the next statement must begin on a new line.

D.1 Basic elements

```
<character> ::= <basic character> | <special character>

<basic character> ::= <space> | [ | . | < | ( | + | & | ] | $ | <star> | ) | ; | - | / | , | _ | > |
                      <question> | : | # | @ | ' | = | <letter> | \ | <digit> | { | } |
                      <vertical bar> |~(tilde) |` (grave)

<special character> ::= ! | ^ | % | "
```

```
<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T
             U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n
             o | p | q | r | s | t | u | v | w | x | y | z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<alphanumeric character> ::= <letter> | <digit> | '
```

D.2 Comments

```
<comment> ::= !<comment character>*
```

D.3 Substitutions

```
<substitution> ::= <variable substitution> |
                   <expression substitution> |
                   <section substitution> |
                   <plane substitution> |
                   <row substitution> |
                   <word substitution> |
                   <column substitution> |
                   <substring substitution> |
                   <length substitution> |
                   <parameter substitution> |
                   <repeated substitution>

<variable substitution> ::= %<assembly-time variable name> |
                            %^<assembly-time variable name> |
                            %<macro variable name> |
                            %^<macro variable name> |
                            %<system variable name>

<system variable name> ::= 'DATE | 'DIM | 'LOGDIM | 'MCOUNT | 'PCOUNT | 'TIME |
                           'TRANSFER

<expression substitution> ::= %=[<assembly-time expression>]

<section substitution> ::= %'SECTION [<data address>]

<plane substitution> ::= %'PLANE [<data address>]

<row substitution> ::= %'ROW [<data address>]

<word substitution> ::= %'WORD [<data address>]

<column substitution> ::= %'COLUMN [<data address>]

<substring substitution> ::= %'SUBSTRING [<selector>]<string ref>
```

```
<selector> ::= <number>? SIZE <number>? |
               UPTO <string> | FROM <string> |
               FROM <string>, UPTO <string>

<string ref> ::= <macro parameter name> |
                 <macro variable name> |
                 <assembly-time variable name> |
                 <number>

<length substitution> ::= %'LENGTH <string ref>

<parameter substitution> ::= %<macro parameter name> |
                             %^<macro parameter name> |
                             %<number> |
                             %^<number> |
                             %'OCCUR [<number>, <string>]

<repeated substitution> ::= %<substitution>
```

<assembly-time variable name>	see D.5
<macro variable name>	see D.6
<assembly-time expression>	see D.4
<row>	see D.15
<column>	see D.15
<number>	see D.4
<string>	see D.4
<macro parameter name>	see D.6

D.4 Assembly-time values

```
<assembly-time-value> ::= [<assembly-time expression>] | <string>

<assembly-time expression> ::= <sign>?<expression>

<expression> ::= <expression><operator><operand> | <operand>

<operand> ::= <assembly-time variable name> |
              <macro variable name> |
              <macro parameter name> |
              <unsigned integer> |
              <hexadecimal value> |
              (<assembly-time expression>)

<operator> ::= + | - | <star> | / | // | 'AND | 'NEQ | 'OR | 'ROTATE | 'SCALE

<string> ::= <string character>*<delimiter>
```

<string character> ::= ˆ<character> | <basic character> | " | ˆ

<delimiter> ::= ! | , |] | <newline>

<number> ::= [<assembly-time expression>] | <unsigned integer>

<numval> ::= <number> | <hexadecimal value>

<sign>	see D.7
<assembly-time variable name>	see D.5
<macro variable name>	see D.6
<macro parameter name>	see D.6
<unsigned integer>	see D.7
<hexadecimal value>	see D.7

D.5 Assembly-time statements

<VAR statement> ::= VAR<assembly-time variable name><preset>?<newline>

<assembly-time variable name> ::= <identifier>

<preset> ::= = <assembly-time value>

<SET statement> ::= SET<assembly-time variable name><preset><newline>

<IF construct> ::= <IF test><choice>FI<newline>

<IF test> ::= IF<condition> | IFN <condition>

<choice> ::= <true part><false part>?

<true part> ::= <APAL construct>*

```
<false part> ::= ELSE <newline><true part> |
                 ELSE_IF <condition><choice> |
                 ELSE_IFN <condition><choice>
```

```
<condition> ::= <arithmetic test><test operand>, <test operand><newline> |
                <monadic string test><string><newline> |
                <diadic string test><string>, <string><newline>
```

<arithmetic test> ::= GT | GE | EQ | LE | LT | NE

<test operand> ::= <number> | <signed integer>

<monadic string test> ::= EX | ID | NU | VA | AV | VS

<diadic string test> ::= BE | SA

```
<LIST statement> ::= LIST <list option><newline>

<list option> ::= FULL | SOURCE | SHORT | NONE

<NOTE statement> ::= NOTE <note type><string><newline>

<note type> ::= TERMINAL | ERROR | WARNING | COMMENT

<STACK statement> ::= STACK <numval><newline>

<APAL construct> ::= <module declaration> |
                     <module header> |
                     <module end> |
                     <global data identity> |
                     DEFINE <newline> |
                     END <newline> |
                     <data section> |
                     <data header> |
                     <data body> |
                     <data end> |
                     <mixed section> |
                     <mixed header> |
                     <code section> |
                     <code header> |
                     <code body> |
                     <code end> |
                     <VAR statement> |
                     <SET statement> |
                     <IF construct> |
                     <LIST statement> |
                     <NOTE statement> |
                     <STACK statement> |
                     <ERASE statement> |
                     <macro definition> |
                     <macro construct> |
                     <macro call>
```

<assembly-time-value>	see D.4
<number>	see D.4
<numval>	see D.4
<signed integer>	see D.7
<string>	see D.4
<module declaration>	see D.9
<module header>	see D.9
<module end>	see D.9
<global data identity>	see D.10

<data section>	see D.11
<data header>	see D.11
<data body>	see D.11
<data end>	see D.11
<mixed section>	see D.12
<mixed header>	see D.12
<code section>	see D.13
<code header>	see D.13
<code body>	see D.13
<code end>	see D.13
<ERASE statement>	see D.6
<macro definition>	see D.6
<macro construct>	see D.6
<macro call>	see D.6

D.6 Macros

```
<macro definition> ::= <macro header><macro body><macro end>

<macro header> ::= MACRO <macro name><parameter template>?<newline>

<macro name> ::= <identifier>

<parameter template> ::= <formal parameter> |
                         <formal parameter>?,<parameter template>

<formal parameter> ::= <macro parameter name><preset>?

<macro parameter name> ::= <identifier>

<macro body> ::= <APAL construct>*

<macro construct> ::= <MEXIT statement> |
                      <MQUIT statement> |
                      <CYCLE construct> |
                      <MVAR statement> |
                      <MSET statement> |
                      <macro comment>

<macro end> ::= MEND <macro name>?<newline>
```

<MEXIT statement> ::= MEXIT <newline>

<MQUIT statement> ::= MQUIT <newline>

<CYCLE construct> ::= <cycle header><cycle body><cycle end>

<cycle header> ::= CYCLE <number>?<newline>

<cycle body> ::= <APAL construct>*<conditional part>?

<conditional part> ::= <cycle test><APAL construct>*

<cycle test> ::=WHILE <condition><newline> |
UNTIL <condition><newline>

<cycle end> ::= REPEAT <newline>

<MVAR statement> ::= MVAR <macro variable name><preset>?<newline>

<macro variable name> ::= <identifier>

<MSET statement> ::=MSET <macro variable name><preset><newline> |
MSET <macro parameter name><preset><newline> |
MSET <number><preset><newline>

<macro comment> ::= !<comment><newline>

<macro call> ::= <macro name><actual parameter list>?<newline>

<actual parameter list> ::= <actual parameter>,<actual parameter list> |
<actual parameter>

<actual parameter> ::= <assembly-time value> |
<macro parameter name> = <assembly-time value>

<ERASE statement> ::= ERASE <macro name><macro name>*<newline>

<preset>	see D.5
<APAL construct>	see D.5
<number>	see D.4
<condition>	see D.5
<comment>	see D.2
<assembly-time value>	see D.4

D.7 Values

<value> ::= <integer value> | <real value> | <character value>

<integer value> ::= <unsigned integer> | <signed integer> | <hexadecimal value>

<unsigned integer> ::= <basic integer> | <basic integer>I<basic integer>

<signed integer> ::= <sign>?<unsigned integer>

<basic integer> ::= <digit><digit>*

<sign> ::= + | -

<hexadecimal value> ::= #<hexadecimal digit><hexadecimal digit>*

<hexadecimal digit> ::= <digit> | A | B | C | D | E | F | a | b | c | d | e | f

```
<real value> ::= <sign>?<basic integer><exponent> |
                 <sign>?<basic integer>.<basic integer>?<exponent>? |
                 <sign>?.<basic integer><exponent>?
```

<exponent> ::= E<sign>?<basic integer>

<character value> ::= "<value character>*"

<value character> ::= ^<character> | <basic character> | !

D.8 Identifiers

<identifier> ::= <letter><alphanumeric character>* | '<alphanumeric character>*

D.9 Modules

<module declaration> ::= <module header><module body>*<module end>

<module header> ::= MODULE <module name><alias>*<newline>

```
<module body> ::= <global data identity> | <data section> | <mixed section> |
                  <code section>
```

<module end> ::= ENDMODULE <module name>?<newline>

<module name> ::= <identifier>

<alias> ::= <identifier>

<global data identity>	see D.10
<data section>	see D.12

<code section>	see D.13
<mixed section>	see D.12

D.10 Data identities

<data identity> ::= <global data identity> | <local data identity>

<global data identity> ::= DEFINE <newline><identity>*END <newline>

<local data identity> ::= <identity>

<identity> ::= <identity name>=<row><data address><newline>

<identity name> ::= <identifier>

<row>	see D.15

D.11 Data sections

<data section> ::= <data header><data body>*<data end>

<data header> ::= DATA <data section name><name property>?<common property>?
<write property>?<newline>

<data section name> ::= <identifier>

<name property> ::= DAP | HOST

<common property> ::= COMMON

<write property> ::= WRITE

<data body> ::= <data declaration><newline> | <length><newline>

<length> := ROWPACK | WORDPACK

<data declaration> ::= <data label>?<data item>*

<data label> ::= <data variable name>:

<data variable name> ::= <identifier>

<data item> ::= <repeat count>?<basic data item> | <repeat count> <data sequence>

<repeat count> ::= <numval><star>

<basic data item> ::= <value><size>? | PLANE | ROW | WORD | ALIGN |
PLANE_ALIGN | ROW_ALIGN

<size> ::= (<numval>)

<data sequence> ::= (<data item><data item>*)

<data end> ::= END <newline>

<numval>	see D.4
<value>	see D.7

D.12 Mixed sections

<mixed section> ::= <mixed header><data body>*<code section>

<mixed header> ::=MIXED <data section name><name property>?<common property>?
<write property>?<newline>

<data body>	see D.11
<code section>	see D.13
<data section name>	see D.11
<name property>	see D.11
<common property>	see D.11
<write property>	see D.11

D.13 Code sections

<code section> ::= <code header><code body>*<code end>

<code header> ::= CODE<code section name><name property>?<newline>

<code section name> ::= <identifier>

```
<code body> ::= <local data identity> |
                <entry point> |
                <TRACE statement> |
                <code label>?<APAL instruction>?<newline>
```

<entry point> ::= ENTRY <entry point name><name property>?<newline>

<entry point name> ::= <identifier>

<code label> ::= <code label name>?:

<code label name> ::= <identifier>

<code end> ::= END <newline>

<name property>	see D.11
<local data identity>	see D.10
<TRACE statement>	see D.14
<APAL instruction>	see appendix F

D.14 Trace statements

```
<trace_statement> ::= TRACE<trace_number>?<registers_trace_item><trace_level><newline> |
                      TRACE<trace_number>?<registers_trace_item>?<trace_level>
                                          <array_store_trace_item><newline>

<array_store_trace_item> ::= <word><modifier>?<trace_count>? WORDPACK?<type/size>?|
                             <word><modifier>?<trace_count>? ROWPACK<type/size>?
                                    <start_bit>?|
                             <word><modifier>?<trace_count>? VERTICAL<type/size>?
                                    <row_range>?<col_range>?|
                             <word><modifier>?<trace_count>? VERTICAL<type/size>?
                                    <col_range>?<row_range>?

<trace_number> ::= <numval>

<registers_trace_item> ::= PER|MER|PER MER|MER PER

<trace_level> ::= LEVEL <numval>

<trace_count> ::= <numval>

<type/size> ::= <type><size>?

<type> ::= HEX|INT|REAL|CHAR|BIT

<size> ::= (<numval>)

<start_bit> ::= FROM_BIT <numval>

<row_range> ::= ROWS (<numval>,<numval>)

<col_range> ::= COLS (<numval>,<numval>)
```

<word>	see D.15
<row>	see D.15
<plane>	see D.15
<modifier>	see D.15
<numval>	see D.4

D.15 Addresses and instruction fields

```
<array store address> ::= <store plane address> | <store row address> |
                          <store column address> | <store word address>

<store plane address> ::= <plane><modifier>?<step>?

<store row address> ::= <row><modifier>?<step A>?

<store column address> ::= <column><modifier>?<step A>?

<store word address> ::= <word><modifier>?<step>?

<data address> ::= <plane> | <row> | <word> | <column>

<plane> ::= <aligned data name><plane offset>? | <plane number>

<name or plane> ::= <data name><plane offset>? | <plane number>

<row> ::= <name or plane><row offset>? | <row offset>

<column> ::= <name or plane><column offset>? | <column offset>

<word> ::= <row> |
           <name or plane>?.<word offset> |
           <name or plane>?<row offset><word offset>

<aligned data name> ::= <data name>

<data name> ::= <data section name> | <data variable name> | <identity name>

<plane offset> ::= + <plane number>

<row offset> ::= .<numval>

<column offset> ::= .<numval>

<word offset> ::= .<numval>

<code store address> ::= <within-section address> | <inter-section address>

<within-section address> ::= <code label name><label offset>? |
                             <star><label offset><doj modifier>?

<inter-section address> ::= <code section name><section offset>?<doj modifier>? |
                            <entry point name><section offset>?<doj modifier>?

<label offset> ::= +<numval> | −<numval>

<section offset> ::= +<numval>

<MCU-or-edge -register bit address> ::= <MCU-or-edge-register>.<bit number><modifier>?<step>?

<modifier> ::= (<mreg>)

<mreg> ::= M1 | M2 | M3 | M4 | M5 | M6 | M7
```

<doj modifier> := (<dojmreg>)

<dojmreg> := M1 | M2 | M3 | M4 | M5 | M6 | M7 | M8 | M9 | M10 | M11 | M12 | M13

<step A> ::= (+A) | (–A) | <step>

<step> ::= (+) | (–)

<modifier>?<step A>? ::= <modifier><step A> | <modifier> | <step A>?

<modifier>?<step>? ::= <modifier><step> | <modifier> | <step>?

<modifier><step A> ::= (<mreg>)(+A) | (<mreg>)(–A) |
(<mreg>+ A) | (<mreg> – A) | <modifier><step>

<modifier><step> ::= (<mreg>)(+) | (<mreg>)(–) | (<mreg> +) | (<mreg>–)

<(inverted) MCU register-1> ::= <(inverted) MCU register>

<(inverted) MCU register-2> ::= <(inverted) MCU register>

<(inverted) MCU register> ::= <inverted MCU register> | <MCU register>

<MCU register-1> ::= <MCU register>

<MCU register-2> ::= <MCU register>

<inverted MCU register> ::= M0N | M1N | M2N | M3N | M4N | M5N | M6N | M7N | M8N |
M9N | M10N | M11N | M12N | M13N

<MCU register> ::= M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7 | M8 | M9 | M10 | M11 | M12 |
M13

<MCU-or-edge-register> := M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7 | M8 | M9 | M10 | M11 | M12 |
M13 | ME

<bit number> ::= <numval>

<direction> ::= R0 | R1 | R2 | R3 | <nesw>

<nesw> ::= N | E | S | W

<geometry> ::= PC | P | CP | C

<count> ::= <numval>

<truth value> ::= T | F | 1 | 0

<times> ::= TIMES

<error number> ::= <numval>

<integer offset> ::= .<numval>

<literal 16> ::= <integer value><size>? | <character value><size>?

<integer value>	see D.7
<character value>	see D.7
<code label name>	see D.13
<code section name>	see D.13
<data section name>	see D.11
<data variable name>	see D.11
<entry point name>	see D.13
<identity name>	see D.10
<numval>	see D.4
<size>	see D.11

Appendix E

Derivation of APAL instruction mnemonics

This appendix contains a number of tables that describe how valid APAL instruction mnemonics are derived, and also how the mnemonics are related to the function of the corresponding APAL instructions.

For array and compound instructions, each letter in the mnemonic has a certain significance, as follows:

Letter	*Meaning*
A	The A plane
B	A single bit of an MCU register or the edge register
C	The C plane
E	Logical equivalence of a bit from an MCU register or the edge register, with each bit of a store plane
F	False (or zero)
I	Inhibit writing to store wherever the corresponding A plane bit is zero. Qualifies S or X when representing the destination of an operation
M	Logical AND of an operand and destination. Qualifies A when representing the destination of an operation
N	Negation (complement) of an operand
O	Orthogonal mode (that is, by columns)
P	Plus
Q	The Q plane

Letter	*Meaning*
R	MCU register, or the edge register
S	Store plane
T	True (or one)
V	Vector add operation
W	Word
X	Store row

You can derive valid APAL mnemonics from the following tables. You can construct a mnemonic from each of the sub-divided boxes by selecting one letter, or sequence of letters, from each compartment, working from left to right.

An asterisk in any compartment means that you can construct valid mnemonics by omitting a choice from that compartment entirely. If a letter appears in parentheses, ( ), it indicates that the letter may optionally be appended onto the letter, or letters, preceding it.

For example:

CPCA is formed by concatenating one item from each compartment in the first box; that is, items **C**, **P**, **C** and **A**

AMEBSN is formed by concatenating one item from each compartment in the large box in centre left of the next page; that is, items **A(M)** and **EBS(N)**

SIPCQS is formed by concatenating one item from each compartment in the box top right of the next page, except the compartment containing an ∗; that is, items **SI**, **P**, **CQ** and **S**

C
CQ
Q
	P	C
		CQ
		Q
			A(N)
			R(N)
			R(N)O
			S(N)

		CQ CQT	

S X	AN F Q R(N)

SI XI	C F Q PCQ

SQ_	AQ CQ QC(N) QF QT

W	F
	R(N)

AMQ AQ	_QQ (N)

C	F Q

C CQ Q	V	CQ

A Q	F T

A(M)
Q
	B(N)
	EBS(N)
	S(N)
	Q
	R(N)
	R(N)O

Q	Q(N)
	A(N)
	C(N)

QT	_CF _AT

R	ANO AX AW W(O) F H(N) QO R(N) S(O) T X(O)

SI	* C CQ Q	P	C CQ Q	S

AS(N)
QA(N)
QC
QF
QT
QS(N)
	_CF

CQ_QQN QS_AS QSN_ASN

DO J(E) J(E)SL NULL SKIP

LOOP PAUSE RLIT RALITR RALITW RAPL RAR RAWD RASC RDGC RAC(E) STOP

ADD(H)(C)
AND
ANDH(N)
DECR
EQV(H)
INCR
MPY(U)32(V)
MPY(U)64
NAND
NANDH(N)
NEQ(H)
NOR
NORH(N)
OR
ORH(N)
SHL(C)
SHR(A)
SHRC
SUB(H)(C)

Appendix F

APAL instructions

This section presents an introduction to the detailed descriptions of individual APAL instructions given later in the appendix. There is also a compact list of mnemonics in appendix A; tables giving the derivation of the mnemonics are presented in appendix E.

APAL code will run on any machine in any range of DAP, whatever its edge size, once the code has been assembled for that size of DAP. The term *ES* is used in this manual (and in other AMT publications) to stand for the edge size of DAP under discussion. Often an example given relates to a machine of particular *ES*, but the principles involved should be clear, and you should not find it difficult to apply them to machines with other values of *ES*.

In this appendix F, most of the instructions are given in alphabetical order, but where an instruction has a companion instruction that performs the same operation on an inverted operand (for example, QS, QSN), or on an orthogonal operand (for example RX, RXO) the two instructions are described together.

Each instruction description has the following general form:

- A heading giving the instruction mnemonic(s)
- A brief description of the function(s) performed by the instruction
- The syntax of the instruction as it is written in an APAL code section.
- The addressing mode, where applicable
- Notes, giving more detail of the syntax or of the function
- Run-time errors that can occur while executing the instruction
- In non-trivial cases, one or more examples

Certain mnemonics are classified as 'pseudo-instructions'; they too are included. One of these, LOOP, generates no code. Many of the others are concerned with loading addresses or values into registers, and differ from other mnemonics in that the instruction type generated is dependent on the size of that address or value. In many cases the value is created in the program literals area, and the assembled instruction references that literal.

In a few specific cases, the actions of two instructions can be performed in a single instruction. Such an instruction is referred to as a 'compound instruction', and its mnemonic is constructed by concatenating the two instruction mnemonics; by convention, the component mnemonics are separated by the '_' character. From a functional point of view, the component instructions are executed in the order as written.

In the description of the operation of the instructions:-

- The complete set of Q registers, for all the PEs, is referred to as the 'Q plane', and similarly for the C and A registers and bits of array store. Thus 'the C plane is copied into the Q plane' is equivalent to 'in every PE the C register is copied into the Q register'
- Where addition operations in the array are described, this means, unless explicitly stated otherwise (for example, in the vector add instructions), that each PE independently performs an add on two or three one-bit operands, producing a Sum and a Carry
- When an MCU register or the edge register is broadcast to the array in 'Main-Store-Mode', it is regarded as having created a fictitious 'R plane', the least significant end of each of whose rows is equal to the register contents, with the most significant end being zero-filled as necessary. Similarly, a broadcast in 'Orthogonal Mode' results in a fictitious 'Orthogonal R plane', the least significant end of each of whose columns is equal to the register contents, with the most significant end being zero-filled as necessary

In the logical operations that can be performed on MCU registers, each AND function is related to one of the NOR functions, in accordance with the usual rules of Boolean algebra; the NAND and OR functions are similarly related. The alternative mnemonics are provided as a programmer convenience.

AB ABN

Function

AB sets every bit of the A plane to a specified bit of an MCU register or the edge register

ABN sets every bit of the A plane to the inverse of a specified bit of an MCU register or the edge register.

Syntax

AB <MCU-or-edge-register>.<bit number><modifier>?<step>?
ABN <MCU-or-edge-register>.<bit number><modifier>?<step>?

where <MCU-or-edge-register>, <bit number>, <modifier>, and <step> together form a register bit address.

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective INT field defines the bit number of the MCU register or the edge register, and the effective ADDR field is discarded.

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction which specifies the register whose bit is addressed by this instruction

2 *bit-number* specifies the value in the INT field of the instruction which is used to construct the effective INT value

3 *modifier* specifies the value in the MOD field of the instruction, which specifies the modifier register to be used, if any. If modifier is omitted the MOD field is set to zero

4 *step* specifies the value in the INCREMENT/DECREMENT field of the instruction, which specifies how the bit number is to be stepped if the instruction appears inside an APAL DO loop.

Possible run-time program errors

None

Example

```
AB  M0.16  (M2)(+)          ! SET EACH BIT OF THE A PLANE TO BIT i OF M0
                            ! WHERE i IS 16 + (INT FIELD OF M2)
                            ! + THE DO LOOP ITERATION NUMBER
```

ADD

ADD

Function

ADD adds two MCU registers, putting the result in one of them. It also assigns the CARRY and OFLO flags.

Syntax

ADD <MCU register-1><MCU register-2>

where

    <MCU register-1> ::= <MCU-register>
    <MCU register-2> ::= <MCU-register>

Notes

1 *MCU-register-1* specifies the value in the MCUR field of the instruction, which specifies both the register containing the first operand and the register to contain the result

2 *MCU-register-2* specifies the value in the MOD field of the instruction which specifies the register containing the second operand

3 The CARRY and OFLO flags are assigned as described in section 1.10

Possible run-time program errors

None.

Example

```
ADD  M3  M5                    ! M3 = M3 + M5
```

ADDC

Function

ADDC adds two MCU registers using the CARRY flag as the carry-in at the least significant bit, putting the result in one of the MCU registers. It also assigns the CARRY and OFLO flags.

Syntax

ADDC <MCU register-1><MCU register-2>

where

<MCU register-1> ::= <MCU-register>
<MCU register-2> ::= <MCU-register>

Notes

1 *MCU-register-1* specifies the value in the MCUR field of the instruction, which specifies both the register containing the first operand and the register to contain the result

2 *MCU-register-2* specifies the value in the MOD field of the instruction which specifies the register containing the second operand

3 The CARRY and OFLO flags are assigned as described in section 1.10

Possible run-time program errors

None.

Example

```
ADD  M2  M10                 ! ADD THE TWO 64-BIT NUMBERS HELD IN
ADDC  M1  M9                 ! REGISTER PAIRS M1, M2 AND M9, M10
```

ADDH ADDH

Function

ADDH adds a literal to an MCU register. It also assigns the CARRY and OFLO flags.

Syntax

ADDH <MCU-register><literal_16>

where <literal_16> see section 6.1.7 for details

Notes

1 *MCU-register* specifies the value in the MCUR field of the instruction which specifies both the register containing the first operand and the register to contain the result

2 *literal_16* specifies the value in the LITERAL field. This value is extended to 32 bits with leading zeros to form the second operand (see section 6.1.7)

3 The CARRY and OFLO flags are assigned as described in section 1.10

Possible run-time program errors

None

Example

```
ADDH  M3  "$"                    ! M3 = M3 + #00002024
```

ADDHC

Function

ADDHC adds a literal to an MCU register using the CARRY flag as the carry-in at the least significant bit. It also assigns the CARRY and OFLO flags.

Syntax

ADDHC <MCU-register><literal_16>

where <literal_16> is defined in section 6.1.7

Notes

1. *MCU-register* specifies the value in the MCUR field of the instruction which specifies both the register containing the first operand and the register to contain the result
2. *literal_16* specifies the value in the LITERAL field. This value is extended to 32 bits with leading zeros to form the second operand (see section 6.1.7)
3. The CARRY and OFLO flags are assigned as described in section 1.10

Possible run-time program errors

None

Example

```
ADDHC  M3  "$"                ! M3 = M3 + #00002024 + CARRY FLAG
```

AEBS AEBSN

AEBS AEBSN

Function

AEBS creates the logical equivalence of each bit of a given store plane with a specified bit of an MCU register or the edge register, putting the result in the A plane.

AEBSN is as AEBS, but uses the inverse of the store plane

Syntax

AEBS <MCU-or-edge-register>.<bit number><plane><modifier>?<step>?
AEBSN <MCU-or-edge-register>.<bit number><plane><modifier>?<step>?

where

<MCU-or-edge-register>, <bit number>, <modifier>, and <step> together form a register bit address

<plane>, <modifier>, and <step> together form a store plane address

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field defines the bit number of the MCU register or the edge register.

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction which specifies the register whose bit is addressed by this instruction

2 *bit-number* specifies the value in the INT field of the instruction which is used to construct the effective INT value

3 *plane* specifies the value in the ADDR field of the instruction, which is used to construct the effective ADDR value

4 *modifier* specifies the value in the MOD field of the instruction, which specifies the modifier register to be used, if any

5 *step* specifies the value in the INCREMENT/DECREMENT field of the instruction, which specifies how both the store plane address and MCU register bit number are to be stepped if the instruction appears inside an APAL DO loop

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
AEBS  M7.0  SPLANE  +  12      ! SET EACH BIT OF THE A PLANE TO
                               ! THE LOGICAL EQUIVALENCE OF
                               ! THE CORRESPONDING BIT OF STORE PLANE
                               ! SPLANE + 12 WITH BIT ZERO OF M7.
                               !
                               !
                               ! SPLANE IS THE NAME OF A DATA
                               ! IDENTITY. BECAUSE THERE
                               ! IS NO MODIFIER FIELD IN
                               ! THE INSTRUCTION, SPLANE + 12
                               ! REPRESENTS AN ABSOLUTE ADDRESS.
```

AF AF

Function

AF sets every bit of the A plane to zero.

Syntax

AF

Possible run-time program errors

None.

AMB AMBN

Function

AMB sets each bit of the A plane to the logical AND of itself and a specified bit of an MCU register or the edge register.

AMBN sets each bit of the A plane to the logical AND of itself and the inverse of a specified bit of an MCU register or the edge register.

Syntax

```
AMB <MCU-or-edge-register>.<bit number><modifier>?<step>?
AMBN <MCU-or-edge-register>.<bit number><modifier>?<step>?
```

where <MCU-or-edge-register>, <bit number>, <modifier>, and <step> together form a register bit address.

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective INT field defines the bit number of the MCU register or the edge register, and the effective ADDR field is discarded.

Notes

See notes for the AB instruction

Possible run-time program errors

None

Example

```
AMB  M2.30(-)                ! SET EACH BIT OF THE A PLANE TO THE
                             ! LOGICAL AND OF ITSELF, WITH BIT i OF M2
                             ! WHERE i IS 30 - THE DO LOOP ITERATION
                             ! NUMBER MODULO 32

AMB  ME.30(-)                ! AS ABOVE, BUT THE EDGE REGISTER IS
                             ! USED, AND THE BIT NUMBER IS MODULO ES
```

AMEBS AMEBSN

AMEBS
AMEBSN

Function

AMEBS creates the logical AND of the A plane with the logical equivalence of each bit of a given store plane with a specified bit of an MCU register or the edge register, putting the result in the A plane.

AMEBSN is as AMEBS, but uses the inverse of the store plane.

Syntax

AMEBS <MCU-or-edge-register>.<bit number><plane><modifier>?<step>?
AMEBSN <MCU-or-edge-register>.<bit number><plane><modifier>?<step>?

where

> <MCU-or-edge-register>, <bit number>, <modifier>, and <step> together form a register bit address
>
> <plane>, <modifier>, and <step> together form a store plane address

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field defines the bit number of the MCU register or the edge register.

Notes

See notes for the AEBS instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
AMEBS  M0.0  SPLANE  (M3)    ! SET EACH BIT OF THE A PLANE
                             ! TO THE LOGICAL AND OF
                             ! ITSELF, WITH THE LOGICAL
                             ! EQUIVALENCE OF
                             ! THE CORRESPONDING BIT OF STORE
                             ! PLANE SPLANE (M3) WITH BIT i OF
                             ! M0, WHERE i IS THE INT FIELD
                             ! OF M3 MODULO 32
```

AMQ (non shifting)
AMQN

Function

AMQ creates the logical AND of the A and Q planes, putting the result in the A plane.

AMQN is as AMQ but uses the inverse of the Q plane.

Syntax

AMQ
AMQN

Notes

1 There is also an AMQ instruction that performs shifting (see next page). The assembler can distinguish between them because the non-shifting AMQ instruction has no operand

Possible run-time program errors

None.

AMQ (shifting) AMQ (shifting)

Function

AMQ (shifting) creates the logical AND of the A and Q planes, putting the result in the A plane, as though the Q plane had first been shifted one place. The Q plane is not changed.

Syntax

AMQ <direction>?<geometry>?<count>?<modifier>
AMQ <nesw><geometry><count>?

where

<direction> ::= <nesw> | R0 | R1 | R2 | R3
<geometry> ::= P | C | PC | CP
<count> ::= <numval>
<nesw> ::= N | E | S | W

Addressing Mode

Mode D addressing is used (see section 7.1.4 for details). The result specifies the effective direction, geometry and count. If the effective count is zero, the instruction is executed as if it were a non-shifting AMQ instruction. Otherwise a shift of one place is performed regardless of the effective count, but the instruction may take longer to execute if the effective count is greater than one.

Notes

1 *direction* or *nesw* specify the value in the DIRECTION field of the instruction, which is used to construct the effective DIRECTION value. If you supply a direction of N, E, S or W, then that value is used at run time as the effective direction regardless of whether or not you supply a modifier. If you specify a modifier then you can either omit the direction, in which case the direction is taken from the modifier at run time, or you can supply a direction of R0, R1, R2 or R3 – meaning a clockwise rotation of 0, 1, 2 or 3 quadrants (each quadrant being 90^{o}) from the direction specified in the modifier. Specifying R0 is identical to omitting the direction

2 *geometry* specifies the value in the GEOMETRY field of the instruction which is used to construct the effective GEOMETRY value. If omitted, the GEOMETRY field is set such that the geometry is taken from the modifier; *modifier* may not be omitted in this case

3 *count* specifies the value in the COUNT field of the instruction, which is used to construct the effective COUNT value. If omitted, zero is assumed

4 *modifier* specifies the value in the MOD field of the instruction, which gives the modifier register to be used, if any

5 The shift of the Q plane is only notional, that is, the Q plane is not itself shifted

Possible run-time program errors

None

Example

```
AMQ  E  P  1            ! PERFORM A NOTIONAL SHIFT OF THE
                        ! Q PLANE ONE PLACE TO THE EAST WITH
                        ! PLANE GEOMETRY, AND 'AND' THE RESULT
                        ! WITH THE A PLANE, PUTTING THE RESULT
                        ! BACK IN THE A PLANE
```

AMQ_QQ

AMQ_QQ

Function

AMQ_QQ is a compound instruction (see section 6.2 for details), and is equivalent to an AMQ (shifting) instruction with a count of one followed by a QQ instruction with a count of one, the instruction pair being executed altogether *n* times where *n* is the effective value of count in the AMQ_QQ instruction.

Syntax

AMQ_QQ <direction>?<geometry>?<count>?<modifier><step>?
AMQ_QQ <nesw><geometry><count>?<step>?

where

<direction> ::= <nesw> | R0 | R1 | R2 | R3
<geometry> ::= P | C | PC | CP
<count> ::= <numval>
<nesw> ::= N | E | S | W

Addressing Mode

Mode D addressing is used (see section 7.1.4 for details). The result specifies the effective direction, geometry and count. If the effective count is zero, the instruction is executed as if it were a non-shifting AMQ instruction.

Notes

See notes 1 to 4 for AMQ (shifting)

Possible run-time program errors

None

Example

```
AMQ_QQ  E  C  10  (M3)        ! THIS INSTRUCTION DOES THE SAME
                              ! AS THE SEQUENCE:
                              !   AMQ E C 1
                              !   QQ E C 1
                              ! EXECUTED (10 + i) TIMES, TAKEN
                              ! MODULO ES, WHERE i IS THE
                              ! COUNT FIELD OF M3.
                              !
                              ! IF (10 + i) MODULO ES IS ZERO
                              ! AMQ_QQ DOES THE SAME AS THE
                              ! SINGLE INSTRUCTION AMQ
```

AMR AMRN

Function

AMR creates the logical AND of the A and R planes, putting the result in the A plane.

AMRN is as AMR, but uses the inverse of the R plane.

Syntax

AMR <MCU-or-edge-register>
AMRN <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction which specifies the register that is used to form the R plane (see section 6.2 for details)

Possible run-time program errors

None.

Example

```
AMR  ME                 ! SET EACH ROW OF THE A PLANE TO THE
                        ! LOGICAL AND OF ITSELF WITH THE
                        ! EDGE REGISTER
```

AMRO AMRNO

Function

AMRO creates the logical AND of the A and orthogonal R planes, putting the result in the A plane.

AMRNO is as AMRO but uses the inverse of the orthogonal R plane.

Syntax

AMRO <MCU-or-edge-register>
AMRNO <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction which specifies the register that is used to form the orthogonal R plane (see section 6.2 for details)

Possible run-time program errors

None.

Example

```
AMRO  ME                    ! SET EACH COLUMN OF THE A PLANE TO THE
                            ! LOGICAL AND OF ITSELF WITH THE
                            ! EDGE REGISTER
```

AMS AMSN

Function

AMS creates the logical AND of the A plane and a given store plane, putting the result in the A plane.

AMSN is as AMS but uses the inverse of the store plane.

Syntax

AMS <plane><modifier>?<step>?
AMSN <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address.

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the AS instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
AMS  SPLANE  (M1)  (+)        ! SET EACH BIT OF THE A PLANE
                              ! TO THE LOGICAL AND OF ITSELF WITH
                              ! THE CORRESPONDING BIT OF STORE
                              ! PLANE SPLANE (M1) + i, WHERE i IS
                              ! THE DO LOOP ITERATION NUMBER
```

AND AND

Function

AND creates the logical AND of two MCU registers, putting the result in one of them. The bits from either or both registers may be inverted before doing the logical AND.

Syntax

AND <(inverted)MCU-register-1><(inverted)MCU-register-2>

where

```
<(inverted)MCU-register-1> ::= <(inverted)MCU-register>
<(inverted)MCU-register-2> ::= <(inverted)MCU-register>
<(inverted)MCU-register> ::= <MCU-register> | <inverted MCU-register>
```

Notes

1 *MCU-register-1* specifies the value in the MCUR field of the instruction, which specifies both the register containing the first operand and the register to contain the result

2 *MCU-register-2* specifies the value in the MOD field of the instruction which specifies the register containing the second operand

Possible run-time program errors

None.

Example

```
AND  M2  M3N                ! M2 = M2 AND (NOT M3)
```

ANDH ANDHN

Function

ANDH creates the logical AND of an MCU register and a literal, putting the result in the register. The bits of the register may be inverted before doing the logical AND.

ANDHN does the same as ANDH but uses the inverse of the literal.

Syntax

ANDH <(inverted)MCU-register><literal_16>
ANDHN <(inverted)MCU-register><literal_16>

where

<(inverted)MCU-register>	see AND instruction
<literal_16>	see section 6.1.7 for details

Notes

1 *MCU-register* specifies the value in the MCUR field of the instruction which specifies both the register containing the first operand and the register to contain the result

2 *literal_16* specifies the value in the LITERAL field of the instruction. This value is extended to 32 bits with leading zeros to form the second operand. In the case of ANDHN this 32-bit literal is inverted before use

Possible run-time program errors

None

Example

```
ANDHN  M1 - 1  (12)            ! M1 = M1 AND #FFFFF000
```

AQ (non-shifting) AQN

Function

AQ copies the Q plane into the A plane.

AQN copies the inverse of the Q plane into the A plane.

Syntax

AQ
AQN

Notes

1 There is also an AQ instruction that performs shifting (see next instruction). The assembler can distinguish between them because the non-shifting AQ instruction has no operands

Possible run-time program errors

None.

AQ (shifting) AQ (shifting)

Function

AQ (shifting) copies the Q plane into the A plane, as though the Q plane had first been shifted one place. The Q plane is not altered.

Syntax

AQ <direction>?<geometry>?<count>?<modifer>
AQ <nesw><geometry><count>?

where

> <direction> ::= <nesw> | R0 | R1 | R2 | R3
> <geometry> ::= P | C | PC | CP
> <count> ::= <numval>
> <nesw> ::= N | E | S | W

Addressing Mode

Mode D addressing is used (see section 7.1.4 for details). The result specifies the effective direction, geometry and count. If the effective count is zero, the instruction is executed as if it were a non-shifting AQ instruction. Otherwise a shift of one place is performed regardless of the effective count, but the instruction may take longer to execute if the effective count is greater than one.

Notes

See notes for the AMQ(shifting)

Possible run-time program errors

None

Example

```
AQ  W  C  3                ! PERFORM A NOTIONAL SHIFT OF THE
                           ! Q PLANE ONE PLACE WEST USING CYCLIC
                           ! AND COPY THE RESULT INTO THE A PLANE
```

AQ_QQ

Function

AQ_QQ is a compound instruction. It first shifts the Q plane the specified number of places, then copies it into the A plane.

Syntax

AQ_QQ <direction>?<geometry>?<count>?<modifier><step>?
AQ_QQ <nesw><geometry><count>?<step>?

where

> <direction> ::= <nesw> | R0 | R1 | R2 | R3
> <geometry> ::= P | C | PC | CP
> <count> ::= <numval>
> <nesw> ::= N | E | S | W

Addressing Mode

Mode D addressing is used (see section 7.1.4 for details). The result specifies the effective direction, geometry and count. If the effective count is zero, the instruction is executed as if it were a non-shifting AQ instruction.

Notes

See notes 1 to 4 for AMQ(shifting)

Possible run-time program errors

None

Example

```
AQ_QQ  E  P  3          ! SHIFT THE Q PLANE THREE PLACES EAST USING
                        ! PLANE GEOMETRY, AND COPY THE RESULT INTO
                        ! BOTH THE Q PLANE AND THE A PLANE
```

AR ARN

Function

AR copies the R plane into the A plane.

ARN copies the inverse of the R plane into the A plane.

Syntax

AR <MCU-or-edge-register>
ARN <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction; it specifies the register used to form the R plane (see section 6.2)

Possible run-time program errors

None.

Example

```
AR  ME                      ! SET EACH ROW OF THE A PLANE EQUAL TO
                            ! THE CONTENTS OF THE EDGE REGISTER
```

ARO ARNO

Function

ARO copies the orthogonal R plane into the A plane.

ARNO copies the inverse of the orthogonal R plane into the A plane.

Syntax

ARO <MCU-or-edge-register>
ARNO <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction; it specifies the register used to form the orthogonal R plane (see section 6.2)

Possible run-time program errors

None.

Example

```
ARO ME                          ! SET EACH COLUMN OF THE A PLANE EQUAL
                                ! TO THE CONTENTS OF THE EDGE REGISTER
```

AS
ASN

Function

AS copies a given store plane into the A plane.

ASN copies the inverse of a given store plane into the A plane.

Syntax

AS <plane><modifier>?<step>?
ASN <plane><modifier>?<step>?

where <plane>, <modifier> and <step> together form a store plane address.

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

1 *plane* specifies the value in the ADDR field of the instruction, which is used to construct the effective ADDR value

2 *modifier* specifies the value in the MOD field of the instruction, which specifies the modifier register to be used, if any

3 *step* specifies the value in the INCREMENT/DECREMENT field of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
AS SPLANE + 1 (M3)  (–)        ! SET EACH BIT OF THE A PLANE
                               ! TO THE CORRESPONDING BIT OF
                               ! PLANE SPLANE (M3) + 1 – i, WHERE i IS
                               ! THE DO LOOP ITERATION NUMBER
```

AS_CF ASN_CF

Function

AS_CF is a compound instruction. It copies a given store plane into the A plane, then sets every bit of the C plane to zero.

ASN_CF copies the inverse of a given store plane into the A plane, then sets every bit of the C plane to zero.

Syntax

AS_CF <plane><modifier>?<step>?
ASN_CF <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address.

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

Same as for AS instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
AS_CF  13  (+)              ! SET EACH BIT OF THE A PLANE TO THE
                            ! CORRESPONDING BIT OF PLANE 13 + i
                            ! WHERE i IS THE DO LOOP ITERATION
                            ! NUMBER.
                            ! THEN SET EACH BIT OF THE C PLANE TO ZERO
```

Function

AT sets every bit of the A plane to one.

Syntax

AT

Possible run-time program errors

None.

CALL CALL

Function

CALL causes a supervisor entry to one of a number of routines that perform privileged operations such as input-output. Generally CALL will be invoked by AMT-supplied macros or subroutines rather than appearing in user programs; operations such as TRACE, STOP and PAUSE are all implemented as particular cases of CALL.

The CALL instruction itself does not change any MCU register or the edge register, though the called routine may do so. A return link is saved in an additional hardware register in the same way as for JESL. This link register is referred to as MP, but it is inaccessible to user code except implicitly through the CALL instruction.

The CALL facilities available and their parameters will be documented elsewhere; contact AMT for further details.

CF CF

Function

CF sets every bit of the C plane to zero.

Syntax

CF

Possible run-time program errors

None.

CPCA CPCAN

Function

CPCA adds corresponding bits of the C and A planes, putting the carry bits in the C plane.

CPCAN is as CPCA but uses the inverse of the A plane.

The sums are discarded.

Syntax

CPCA
CPCAN

Possible run-time program errors

None.

CPCQ CPCQ

Function

CPCQ adds corresponding bits of the C and Q planes, putting the carry bits in the C plane. The sums are discarded.

Syntax

CPCQ

Possible run-time program errors

None.

CPCQA
CPCQAN

Function

CPCQA adds corresponding bits of the C, Q and A planes, putting the carry bits in the C plane.

CPCQAN is as CPCQA but uses the inverse of the A plane.

The sums are discarded.

Syntax

CPCQA
CPCQAN

Possible run-time program errors

None.

CPCQR CPCQRN

Function

CPCQR adds corresponding bits of the C, Q and R planes, putting the carry bits in the C plane.

CPCQRN is as CPCQR but uses the inverse of the R plane.

The sums are discarded.

Syntax

CPCQR <MCU-or-edge-register>
CPCQRN <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction; it specifies the register used to form the R plane (see section 6.2)

Possible run-time program errors

None.

CPCQRO CPCQRNO

Function

CPCQRO adds corresponding bits of the C, Q and orthogonal R planes, putting the carry bits in the C plane.

CPCQRNO is as CPCQRO but uses the inverse of the orthogonal R plane.

The sums are discarded.

Syntax

CPCQRO <MCU-or-edge-register>
CPCQRNO <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction; it specifies the register used to form the orthogonal R plane (see section 6.2)

Possible run-time program errors

None.

CPCQS CPCQSN

Function

CPCQS adds corresponding bits of the C and Q planes and a given store plane putting the carry bits in the C plane.

CPCQSN is as CPCQS but uses the inverse of the store plane.

The sums are discarded.

Syntax

CPCQS <plane><modifier>?<step>?
CPCQSN <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address.

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

1 *plane* specifies the value in the ADDR field of the instruction, which is used to construct the effective ADDR value

2 *modifier* specifies the value in the MOD field of the instruction, which specifies the modifier register to be used, if any

3 *step* specifies the value in the INCREMENT/DECREMENT field of the instruction, which specifies how the store plane adfi ! Adobe PostScript(tm) via Sun Microsystems PC-NFS dress is to APAL DO loop

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
CPCQS  28  (M4)              ! SET EACH BIT OF THE C PLANE TO THE
                             ! CARRY BIT GENERATED BY ADDING THAT
                             ! C PLANE BIT TO THE CORRESPONDING BITS
                             ! OF THE Q PLANE AND
                             ! STORE PLANE 28 + (ADDR FIELD OF M4)
```

CPCQT CPCQT

Function

CPCQT adds corresponding bits of the C and Q planes and a notional plane consisting of all ones, putting the carry bits in the C plane. The sums are discarded.

Syntax

CPCQT

Possible run-time program errors

None.

CPCR CPCRN

Function

CPCR adds corresponding bits of the C and R planes putting the carry bits in the C plane.

CPCRN is as CPCR but uses the inverse of the R plane.

The sums are discarded.

Syntax

CPCR <MCU-or-edge-register>
CPCRN <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction; it specifies the register used to form the R plane (see section 6.2)

Possible run-time program errors

None.

CPCRO CPCRNO

CPCRO
CPCRNO

Function

CPCRO adds corresponding bits of the C and orthogonal R planes, putting the carry bits in the C plane.

CPCRNO is as CPCRO but uses the inverse of the orthogonal R plane.

The sums are discarded.

Syntax

CPCRO <MCU-or-edge-register>
CPCRNO <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction; it specifies the register used to form the orthogonal R plane (see section 6.2)

Possible run-time program errors

None.

CPCS
CPCSN

Function

CPCS adds corresponding bits of the C plane and a given store plane, putting the carry bits in the C plane.

CPCSN is as CPCS but uses the inverse of the store plane.

The sums are discarded.

Syntax

CPCS <plane><modifier>?<step>?
CPCSN <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address.

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the CPCQS instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
CPCS  SPLANE  (M2)          ! SET EACH BIT OF THE C PLANE TO THE
                            ! CARRY BIT GENERATED BY ADDING
                            ! THAT C PLANE BIT TO THE CORRESPONDING
                            ! BIT OF STORE PLANE SPLANE (M2)
```

CPQA CPQAN

Function

CPQA adds corresponding bits of the Q and A planes, putting the carry bits in the C plane.

CPQAN is as CPQA but uses the inverse of the A plane.

The sums are discarded.

Syntax

CPQA
CPQAN

Possible run-time program errors

None.

CPQR CPQRN

Function

CPQR adds corresponding bits of the Q and R planes, putting the carry bits in the C plane.

CPQRN is as CPQR but uses the inverse of the R plane.

The sums are discarded.

Syntax

CPQR <MCU-or-edge-register>
CPQRN <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction; it specifies the register used to form the R plane (see section 6.2)

Possible run-time program errors

None.

CPQRO CPQRNO

Function

CPQRO adds corresponding bits of the Q and orthogonal R planes, putting the carry bits in the C plane.

CPQRNO is as CPQRO but uses the inverse of the orthogonal R plane.

The sums are discarded.

Syntax

CPQRO <MCU-or-edge-register>
CPQRNO <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction; it specifies the register used to form the orthogonal R plane (see section 6.2)

Possible run-time program errors

None.

CPQS CPQSN

Function

CPQS adds corresponding bits of the Q plane and a given store plane, putting the carry bits in the C plane.

CPQSN is as CPQS but uses the inverse of the store plane.

The sums are discarded.

Syntax

CPQS <plane><modifier>?<step>?
CPQSN <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address.

Notes

See notes for the CPCQS instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
CPQS SPLANE  (M4)  (+)        ! SET EACH BIT OF THE C PLANE
                              ! TO THE CARRY BIT GENERATED BY
                              ! ADDING CORRESPONDING BITS
                              ! OF THE Q PLANE AND STORE PLANE
                              ! SPLANE (M4) + i, WHERE i IS
                              ! THE DO LOOP ITERATION NUMBER
```

CQ CQ

Function

CQ copies the Q plane into the C plane.

Syntax

CQ

Possible run-time program errors

None.

CQPCA
CQPCAN

Function

CQPCA adds corresponding bits of the C and A planes, putting the carry bits in the C plane and the sums in the Q plane.

CQPCAN is as CQPCA but uses the inverse of the A plane.

Syntax

CQPCA
CQPCAN

Possible run-time program errors

None.

CQPCQ CQPCQ

Function CQPCQ adds corresponding bits of the C and Q planes, putting the carry bits in the C plane and the sums in the Q plane.

Syntax

CQPCQ

Possible run-time program errors

None.

CQPCQA CQPCQAN

Function

CQPCQA adds corresponding bits of the C, Q and A planes, putting the carry bits in the C plane and the sums in the Q plane.

CQPCQAN is as CQPCQA but uses the inverse of the A plane.

Syntax

CQPCQA
CQPCQAN

Possible run-time program errors

None.

CQPCQR CQPCQRN

Function

CQPCQR adds corresponding bits of the C, Q and R planes, putting the carry bits in the C plane and the sums in the Q plane.

CQPCQRN is as CQPCQR but uses the inverse of the R plane.

Syntax

CQPCQR <MCU-or-edge-register>
CQPCQRN <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction; it specifies the register used to form the R plane (see section 6.2)

Possible run-time program errors

None.

CQPCQRO
CQPCQRNO

Function

CQPCQRO adds corresponding bits of the C, Q and orthogonal R planes, putting the carry bits in the C plane and the sums in the Q plane.

CQPCQRNO is as CQPCQRO but uses the inverse of the orthogonal R plane.

Syntax

CQPCQRO <MCU-or-edge-register>
CQPCQRNO <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction; it specifies the register used to form the orthogonal R plane (see section 6.2)

Possible run-time program errors

None.

CQPCQS CQPCQSN

Function

CQPCQS adds corresponding bits of the C and Q planes and a given store plane putting the carry bits in the C plane and the sums in the Q plane.

CQPCQSN is as CQPCQS but uses the inverse of the store plane.

Syntax

CQPCQS <plane><modifier>?<step>?
CQPCQSN <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address.

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the CPCQS instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
CQPCQS 26 (M6) (+)      ! SET EACH BIT OF THE Q PLANE
                        ! TO THE SUM OF THAT BIT,
                        ! THE CORRESPONDING BIT OF THE C PLANE
                        ! AND THE CORRESPONDING BIT OF
                        ! STORE PLANE 26 + (ADDR FIELD OF M6) + i
                        ! WHERE i IS THE DO LOOP ITERATION
                        ! NUMBER.
                        !
                        ! THE CARRY BIT IS PLACED IN THE
                        ! CORRESPONDING BIT OF THE C PLANE.
```

CQPCQT CQPCQT

Function

CQPCQT adds corresponding bits of the C and Q planes and a notional plane consisting of all ones, putting the carry bits in the C plane and the sums in the Q plane.

Syntax

CQPCQT

Possible run-time program errors

None.

CQPCR CQPCRN

Function

CQPCR adds corresponding bits of the C and R planes, putting the carry bits in the C plane and the sums in the Q plane.

CQPCRN is as CQPCR but uses the inverse of the R plane.

Syntax

CQPCR <MCU-or-edge-register>
CQPCRN <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction; it specifies the register used to form the R plane (see section 6.2)

Possible run-time program errors

None.

CQPCRO CQPCRNO

Function

CQPCRO adds corresponding bits of the C and orthogonal R planes, putting the carry bits in the C plane and the sums in the Q plane.

CQPCRNO is as CQPCRO but uses the inverse of the orthogonal R plane.

Syntax

CQPCRO <MCU-or-edge-register>
CQPCRNO <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction; it specifies the register used to form the orthogonal R plane (see section 6.2)

Possible run-time program errors

None.

CQPCS CQPCSN

Function

CQPCS adds corresponding bits of the C plane and a given store plane, putting the carry bits in the C plane and the sums in the Q plane.

CQPCSN is as CQPCS but uses the inverse of the store plane.

Syntax

CQPCS <plane><modifier>?<step>?
CQPCSN <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address.

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the CPCQS instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
CQPCS SPLANE  (M4)  (-)        ! SET EACH BIT OF THE Q PLANE
                               ! TO THE SUM OF THE CORRESPONDING
                               ! BITS OF THE C PLANE AND
                               ! STORE PLANE SPLANE (M4) - i
                               ! WHERE i IS THE DO LOOP ITERATION
                               ! NUMBER.
                               !
                               ! THE CARRY BIT IS PLACED IN
                               ! THE CORRESPONDING BIT OF
                               ! THE C PLANE.
```

CQPQA CQPQAN

Function

CQPQA adds corresponding bits of the Q and A planes, putting the carry bits in the C plane and the sums in the Q plane.

CQPQAN is as CQPQA but uses the inverse of the A plane.

Syntax

CQPQA
CQPQAN

Possible run-time program errors

None.

CQPQR CQPQRN

CQPQR CQPQRN

Function

CQPQR adds corresponding bits of the Q and R planes, putting the carry bits in the C plane and the sums in the Q plane.

CQPQRN is as CQPQR but uses the inverse of the R plane.

Syntax

CQPQR <MCU-or-edge-register>
CQPQRN <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction; it specifies the register used to form the R plane (see section 6.2)

Possible run-time program errors

None.

CQPQRO
CQPQRNO

Function

CQPQRO adds corresponding bits of the Q and orthogonal R planes, putting the carry bits in the C plane and the sums in the Q plane.

CQPQRNO is as CQPQRO but uses the inverse of the orthogonal R plane.

Syntax

CQPQRO <MCU-or-edge-register>
CQPQRNO <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction; it specifies the register used to form the orthogonal R plane (see section 6.2)

Possible run-time program errors

None.

CQPQS CQPQSN

Function

CQPQS adds corresponding bits of the Q plane and a given store plane, putting the carry bits in the C plane and the sums in the Q plane.

CQPQSN is as CQPQS but uses the inverse of the store plane.

Syntax

CQPQS <plane><modifier>?<step>?
CQPQSN <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the CPCQS instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
CQPQS  23  (M5)  (+)          ! SET EACH BIT OF THE Q PLANE TO
                              ! THE SUM OF THAT BIT AND
                              ! THE CORRESPONDING BIT OF STORE
                              ! PLANE 23 + (ADDR FIELD OF M5) + i
                              ! WHERE i IS THE DO LOOP ITERATION
                              ! NUMBER.
                              !
                              ! THE CARRY BIT IS PLACED IN
                              ! THE CORRESPONDING BIT OF THE C PLANE.
```

CQ_QQN CQ_QQN

Function

CQ_QQN is a compound instruction. It copies the Q plane into the C plane, then inverts every bit of the Q plane.

Syntax

CQ_QQN

Possible run-time program errors

None.

CQVCQ CQVCQ

Function

CQVCQ adds together corresponding rows or columns of the C and Q planes treating each pair of rows or columns as *ES*-sized unsigned integers. The results are placed in the Q plane, the carry-out bits in the C plane.

Syntax

CQVCQ <direction>?<geometry>?<count>?<modifier><step>?
CQVCQ <nesw><geometry><count>?<step>?

where

<direction> ::= <nesw> | R0 | R1 | R2 | R3
<geometry> ::= P | C | PC | CP
<count> ::= <numval>
<nesw> ::= N | E | S | W

Addressing Mode

Mode D addressing is used (see section 7.1.4 for details). The result specifies the effective direction, geometry and count.

Notes

1 *direction* or *nesw* specify the value in the DIRECTION field of the instruction, which is used to construct the effective DIRECTION value. If omitted the value R0 is assumed; *modifier* is mandatory in this case

2 *geometry* specifies the value in the GEOMETRY field of the instruction which is used to construct the effective GEOMETRY value. If omitted, the GEOMETRY field is set such that the geometry is specified by the modifier; *modifier* is mandatory in this case

3 *count* specifies the value in the COUNT field of the instruction, which is used to construct the effective COUNT value. If omitted, zero is assumed. Henceforth, in the notes for this instruction, the effective COUNT value is denoted by EC

4 *modifier* specifies the value in the MOD field of the instruction which indicates which modifier register is to be used, if any

5 The direction of carry propagation is given by the effective DIRECTION value. This determines how the Q and C planes are to be added (as *ES* pairs of either rows or columns), and the ordering of the bits in the rows or columns (since the direction of carry propagation is that of increasing bit significance)

5 (continued)

effective DIRECTION	*form of addition*
North	Add corresponding pairs of columns; carry bits propagate northwards
South	Add corresponding pairs of columns; carry bits propagate southwards
East	Add corresponding pairs of rows; carry bits propagate eastwards
West	Add corresponding pairs of rows; carry bits propagate westwards

6 The geometry of carry propagation is given by the effective GEOMETRY value. Hence the carry-in bit(s) at the least significant end are either all zeros (for PLANE geometry) or the bits carried out from the most significant end (for CYCLIC geometry)

7 For CQVCQ and CVCQ (covered next in this appendix) the C plane receives the current carry-out bits. For CQVCQ and QVCQ (covered later in this appendix) the Q plane receives the current sum bits

8 The addition of a pair of rows or columns can be regarded as ES separate additions; each of these takes as operands one bit from the original Q plane, the corresponding bit of the original C plane, and the carry-out from adding the next less significant bits.

In each row or column the carry-out bits are only known initially at certain positions called *defined carry points*. These are:

a In plane geometry, the conceptual bit position outside the least significant edge

b In either geometry, any bit position where corresponding initial bits of the C and Q planes are the same

At the end of the instruction, sum bits are only guaranteed in the EC + 1 places more significant than any defined carry point. Carry-out bits are defined in these places and also at the defined carry points. Thus when EC has its maximum value of $ES - 1$, this guarantees ES defined sum and carry-out bits in every row or column if plane geometry is specified. With cyclic geometry, if the Q plane is the inverse of the C plane, there are no defined carry points and thus no defined sum or carry-out bits, whatever the value of EC.

Note that specific hardware implementations (including the initial versions of DAP 500 and DAP 600) may ignore the value of EC and always allow time for carrys to propagate along the entire row or column. However, for compatibility with possible future implementations, you are recommended to specify *count* as described above. It is convenient to use a value of $ES - 1$, except for cases where the performance of CQVCQ is critical

CQVCQ CQVCQ

Possible run-time program errors

None

Example

This example is from code to run on a DAP 500; the example assumes the following user-defined convention for the data in each row of the Q and C planes:

There are two data fields, in bits 8 to 14 and 24 to 31 of each row. Bit 15 is always zero.

bit positions	*contents*
0 - 7	undefined
8 - 14	data field, 7 bits
15	0
16 - 23	undefined
24 - 31	data field, 8 bits

The following instruction is used:

```
CQVCQ W P 7
```

The instruction has an effective DIRECTION of west, an effective GEOMETRY of plane and an effective COUNT of 7 (EC = 7). Bit 15 is a defined carry point; because the geometry is plane, notional bit 32 is also a defined carry point. EC = 7 guarantees 8 defined sum bits in the 8 positions more significant than any defined carry point, and 9 defined carry-out bits in the same positions also including the defined carry point. However, the results in any position can only be defined if the original data was defined. The resulting sum and carry bits are therefore as follows:

bit positions	*sum bits*	*carry-out bits*
0 - 7	undefined	undefined
8 - 14	7-bit data sum	defined
15	undefined	defined (0)
16 - 23	undefined	undefined
24 - 31	8-bit data sum	defined

The defined carry-out bits in positions 8 and 24 indicate whether there was overflow in the 7-bit or 8-bit data sums respectively.

CVCQ

Function

CVCQ adds together corresponding rows or columns of the C and Q planes treating each pair of rows or columns as *ES*-sized unsigned integers. The carry-out bits are placed in the C plane; the Q plane is unchanged by this instruction.

Syntax

CVCQ <direction>?<geometry>?<count>?<modifier><step>?
CVCQ <nesw><geometry><count>?<step>?

where

<direction> ::= <nesw> | R0 | R1 | R2 | R3
<geometry> ::= P | C | PC | CP
<count> ::= <numval>
<nesw> ::= N | E | S | W

Addressing Mode

Mode D addressing is used (see section 7.1.4 for details). The result specifies the effective direction, geometry and count.

Notes

See notes for the CQVCQ instruction

Possible run-time program errors

None.

Example

```
CVCQ  W  P  31
```

In this DAP 600 example the Q and C planes are added as 64 rows of integers. Notional bit position 64 is a defined carry point because of plane geometry. A count of 31 guarantees 32 result sum and carry-out bits in positions 32 to 63 of each row. The carry-out bits go into the C plane; the Q plane is unchanged.

DECR

Function

DECR subtracts one from a specified MCU register. It also assigns the CARRY and OFLO flags.

Syntax

DECR <MCU-register>

Notes

1 *MCU-register* specifies the value in the MCUR field of the instruction which specifies the register whose value is to be decremented

2 The value in the specified MCU register is treated as a 32-bit unsigned integer. The CARRY and OFLO flags are assigned as described in section 1.10

Possible run-time program errors

None.

Example

```
DECR  M7                ! M7 = M7 - 1
                        ! SET THE CARRY FLAG UNLESS THE ORIGINAL M7 = 0
```

DO DO

Function

DO indicates the start of an *APAL DO loop*, and also specifies how many times the instruction sequence within the loop is to be executed.

An APAL DO loop is a sequence of up to 256 APAL instructions, excluding the DO instruction itself, that is to be executed repeatedly the number of times specified in the DO instruction (see the syntax description below). The first instruction in the loop is the instruction that immediately follows the DO instruction.

A DO loop is terminated by whichever of the following occurs first:

- The pseudo-instruction LOOP; the instruction preceding LOOP is the last instruction in the loop. LOOP itself generates no code; it is merely an indicator of the end of a DO loop
- A labelled instruction; that instruction is the last instruction in the loop. A null label, consisting of a colon with no preceding identifier, can be used for this purpose. Note that the label need not occur on the same source line as the instruction that it labels; it can be separated from the instruction by comment lines, or by lines containing only labels.

The assembler counts the number of instructions between the DO and the terminating instruction (not including the DO or a LOOP, but including the labelled instruction), and stores that number in the machine code version of the DO.

For example the following are all equivalent, the AMS instruction being the last instruction in the loop in every case:

```
      DO 32 TIMES    DO 32 TIMES    DO 32 TIMES         DO 32 TIMES
      QA             QA             QA                  QA
      AMS 0 (M1+)   :              :                LAST : AMS 0 (M1+)
      LOOP           AMS 0 (M1+)    ! NEXT IS LAST
                                    AMS 0 (M1+)

      DO 32 TIMES                   DO 32 TIMES
      QA                            QA
     :                             : AMS 0 (M1+)
LAST : AMS 0(M1+)
```

The instructions inside an APAL DO loop (that is, after the DO instruction and up to and including the last instruction) can be any of those described in this appendix except DO, JSL, JESL or STOP. In particular, this implies that APAL DO loops cannot be nested. The last instruction in a DO loop must not be a SKIP. The instructions have the same effect as if they were not in a DO loop, except that the INCREMENT/DECREMENT field can cause stepping of either or both of store addresses and MCU or edge register bit numbers on each execution of the loop (see section 7.1 for details). The J, JE or EXIT instructions will prematurely terminate the DO loop.

DO

DO

Syntax

DO <DO count><doj modifier>?<times>?
DO <doj modifier><times>?

where

> <DO count> ::= <numval>
> <doj modifier> ::= (<dojmreg>)
> <dojmreg> := M1 | M2 | M3 | M4 | M5 | M6 | M7 | M8 | M9 | M10 | M11 | M12 | M13
> <times> ::= TIMES

Addressing Mode

Mode E addressing is used (see section 7.1.5 for details).

Notes

1 *DO count* specifies the number of times that the DO loop will be executed and can be modified. If *DO count* is zero, *modifier* is obligatory

2 *modifier* specifies the value in the MOD field, which gives the modifier register to be used, if any

3 The length of the DO loop is encoded in the DO instruction, and has a maximum value of 256 instructions. Registers M1 to M13 are available as modifiers for this instruction

Possible run-time program errors

A run-time program error will occur if an attempt is made to execute a DO instruction inside another DO loop.

Examples

```
 DO  32  TIMES              ! EXECUTE THE LOOP 32 TIMES.
 QS  0  (M2)  (+)           ! THESE STORE PLANE ADDRESSES ARE INCREASED
:SQ  0  (M3)  (+)           ! BY ONE PLANE EACH TIME THE LOOP IS
                            ! EXECUTED
                            !
 DO  0  (M1)  TIMES         ! EXECUTE THE LOOP i TIMES, WHERE i IS
:SF  31  (M2)  (-)          ! THE VALUE IN M1
```

EQV

Function

EQV creates the logical equivalence of two MCU registers, putting the result in one of them.

Syntax

EQV <MCU-register-1><MCU-register-2>

where

    <MCU-register-1> ::= <MCU-register>
    <MCU-register-2> ::= <MCU-register>

Notes

1 *MCU-register-1* specifies the value in the MCUR field of the instruction, which specifies both the register containing the first operand and the register to contain the result

2 *MCU-register-2* specifies the value in the MOD field of the instruction which specifies the register containing the second operand

Possible run-time program errors

None.

Example

```
EQV  M3  M4                   ! SET EACH BIT OF M3 TO THE LOGICAL EQUIVALENCE
                              ! OF CORRESPONDING BITS OF M3 AND M4.
```

EQVH

Function

EQVH creates the logical equivalence of an MCU register and a literal, putting the result in the register.

Syntax

EQVH <MCU-register><literal_16>

where <literal_16> is defined in section 6.1.7

Notes

1 *MCU-register* specifies the value in the MCUR field of the instruction which specifies both the register containing the first operand and the register to hold the result

2 *literal_16* specifies the value in the LITERAL field of the instruction. This value is extended to 32 bits with leading zeros to form the second operand

Possible run-time program errors

None

Example

```
EQVH  M1  31                ! SET EACH BIT OF M1 TO THE LOGICAL
                            ! EQUIVALENCE OF THE CORRESPONDING
                            ! BITS OF M1 WITH #0000001F.
```

EXIT

Function

EXIT loads an instruction address into the program counter using the value in a specified MCU register, thereby causing a transfer of control.

EXIT is the normal way of returning control from a subroutine to the point from which the subroutine was entered (via a JSL or JESL instruction). Software conventions to be used when writing APAL subroutines are given in Chapter 9.

Syntax

EXIT <MCU-register>?<offset>?

where <offset> ::= <numval>

Addressing Mode

Mode F addressing is used (see section 7.1.6 for details). The result specifies the address of the destination instruction.

Notes

1 *MCU-register* specifies the value in the MCUR field of the instruction. This specifies the register from whose contents is derived the value that is loaded into the program counter. If *MCU-register* is omitted, then M0 is used by default

2 *offset* specifies the value in the CODE ADDRESS field of the instruction, and represents an optional offset from the value in the register. If *offset* is omitted, zero is assumed

3 The transfer of control between APAL and FORTRAN-PLUS and/or an associated host program is described in chapter 9

Possible run-time program errors

A run-time error will occur if the address of the destination instruction is outside the range defined by the DAP program block.

Examples

```
EXIT                    ! PROGRAM COUNTER = THE LEAST SIGNIFICANT
                        ! 20 BITS OF M0

EXIT  40                ! PROGRAM COUNTER = THE LEAST SIGNIFICANT
                        ! 20 BITS OF (M0 + 40)
```

INCR

INCR

Function

INCR adds one to an MCU register. It also assigns the CARRY and OFLO flags.

Syntax

INCR <MCU-register>

Notes

1 *MCU-register* specifies the value in the MCUR field of the instruction which specifies the register whose value is to be incremented

2 The CARRY and OFLO flags are assigned as described in section 1.10

Possible run-time program errors

None.

Example

```
INCR  M6                  ! M6 = M6 + 1; RESET THE CARRY FLAG
                          ! UNLESS ORIGINAL M6 = -1 (32)
```

J

Function

J loads an instruction address into the program counter, thereby causing control to be transferred to that instruction. The instruction to which control is transferred must be in the same code section as the J instruction.

Syntax

J <code label name><label offset>?<doj modifier>?
J <star><label offset><doj modifier>?

where <code label name> (or <star>) and <label offset> together form a within-section instruction address as described in section 7.3.5

Addressing Mode

If a modifier is specified, then mode F addressing is used (see section 7.1.6 for details), the result giving the destination instruction address.

Notes

1. The instruction address specifies the value in the CODE ADDRESS field of the instruction, which is the value that will be loaded into the program counter
2. An error will occur during consolidation if the instruction address is not in the same code section as the J instruction

Possible run-time program errors

A run-time error will occur if the address of the destination instruction is outside the range defined by the DAP program block.

Examples

```
J LAB               ! TRANSFER CONTROL TO THE INSTRUCTION
                    ! LABELLED LAB

J LAB + 4           ! TRANSFER CONTROL TO THE FOURTH
                    ! INSTRUCTION AFTER THE INSTRUCTION
                    ! LABELLED LAB

J LAB(M3)           ! TRANSFER CONTROL TO THE INSTRUCTION
                    ! n INSTRUCTIONS AFTER THE INSTRUCTION
                    ! LABELLED LAB, WHERE n IS THE CONTENTS
                    ! OF REGISTER M3

J * - 12            ! TRANSFER CONTROL TO THE TWELFTH
                    ! INSTRUCTION BEFORE THE J INSTRUCTION
```

JE JE

Function

JE loads an instruction address into the program counter, thereby causing control to be transferred to that instruction. The instruction to which control is transferred is identified relative to the first instruction of a named entry point in the same or another code section.

Syntax

JE <code section name><section offset>?<doj modifier>?
JE <entry point name><section offset><doj modifier>?

where <code section name> (or <entry point name>) and <section offset> together form an inter-section instruction address as described in section 7.3.5

Addressing Mode

If a modifier is specified, then mode F addressing is used (see section 7.1.6 for details), the result giving the destination instruction address.

Notes

1 The instruction address specifies the value in the CODE ADDRESS field of the instruction, which is the value that will be loaded into the program counter

2 When transferring control between code sections, you should be aware of the code section conventions described in chapter 9.

Possible run-time program errors

A run-time error will occur if the address of the destination instruction is outside the range defined by the DAP program block.

Examples

```
JE  SECTION3 + 9        ! TRANSFER CONTROL TO THE 10TH
                        ! INSTRUCTION OF CODE SECTION 'SECTION3'

JE  EP2                 ! TRANSFER CONTROL TO THE INSTRUCTION
                        ! AT ENTRY POINT EP2.
```

JESL   JESL

Function

JESL transfers control to a named code section or entry point (see the JE instruction) but before loading the appropriate instruction address into the program counter, the current value of the program counter +1 is saved in a specified MCU register. This value can be used by a subsequent EXIT to return control to the instruction following the JESL.

The JESL or JSL instructions are the normal means of calling a subroutine. Software conventions to be used when writing APAL subroutines are given in Chapter 9.

Syntax

JESL <MCU-register>?<code section name><section offset>?
JESL <MCU-register>?<entry point name><section offset>?

where <code section name> (or <entry point name>) and <section offset> together form an inter-section instruction address as described in section 7.3.5.

Notes

1 *MCU-register* specifies the value in the MCUR field of the instruction which specifies the register in which the current program counter value +1 is to be saved. This value is held in the least significant 20 bits of *MCU-register* ; all remaining bits are undefined.

  If *MCU-register* is omitted, the current program counter value +1 is saved in M0 by default

2 The instruction address specifies the value in the CODE ADDRESS field of the instruction, which is the value that will be loaded into the program counter

3 When transferring control between code sections, you should be aware of the code section conventions described in chapter 9.

Possible run-time program errors

A run-time program error will occur if the address in the J field is outside the DAP program block, or if a JESL instruction is encountered in a DO loop.

Example

```
JESL  SECTION3  +  9          ! AS FOR JE, EXCEPT THAT THE CURRENT VALUE
                              ! OF THE PROGRAM COUNTER +1 IS SAVED IN M0
```

JSL JSL

Function

JSL transfers control to another instruction in the same code section (see the J instruction), but before loading the appropriate instruction address into the program counter the current value of the program counter +1 is saved in a specified MCU register. The value can be used in a subsequent EXIT to return control to the instruction following the JSL.

The JSL or JESL instructions are the normal means of calling a subroutine.

Syntax

```
JSL <MCU-register>?<code label name><label offset>?
JSL <MCU-register>?<star><label offset>?
```

where <code label name> (or <star>) and <label offset> together form a within-section instruction address as described in section 7.3.5.

Notes

1 *MCU-register* specifies the value in the MCUR field of the instruction which specifies the register in which the current program counter value +1 is to be saved. This value is held in the least significant 20 bits of *MCU-register*; all remaining bits are undefined.

  If *MCU-register* is omitted, M0 is assumed by default

2 The instruction address specifies the value in the CODE ADDRESS field of the instruction, which specifies the value that will be loaded into the program counter

3 An error will occur during consolidation if the instruction address is not in the same code section as the JSL instruction

Possible run-time program errors

A run-time program error will occur if the address in the J field is outside the DAP program block, or a JSL instruction is encountered in a DO loop.

Example

```
JSL  LAB                    ! AS FOR J EXCEPT THAT THE CURRENT VALUE
                            ! OF THE PROGRAM COUNTER +1 IS SAVED IN M0
```

LOOP LOOP

Function

LOOP is a pseudo instruction that indicates the end of an APAL DO LOOP (see the DO instruction)

LOOP generates no code and should not be labelled.

Syntax

LOOP

MPY32 MPY32

Function

MPY32 performs a multiplication of the contents of two MCU registers, regarded as 32-bit signed integers. The least significant half of the 64-bit product is written to the first MCU register. The most significant half is discarded.

Syntax

MPY32 <MCU-register-1><MCU-register-2>

where

<MCU-register-1> ::= <MCU-register>
<MCU-register-2> ::= <MCU-register>

Notes

1 *MCU-register-1* states the value to be placed in the MCUR field of the instruction, specifying the register which when the instruction is entered will hold the first operand and when the instruction has been completed will hold the least significant half of the product

2 *MCU-register-2* states the value to be placed in the MOD field of the instruction, specifying the register containing the second operand

Possible run-time program errors

None

Example

```
MPY32  M3  M7              ! FORM THE PRODUCT OF THE CONTENTS
                           ! OF M3 AND M7, PUT THE LEAST
                           ! SIGNIFICANT HALF BACK IN M3 AND
                           ! DISCARD THE MOST SIGNIFICANT HALF
```

MPY32V MPY32V

Function

MPY32V performs a multiplication of the contents of two MCU registers, regarded as 32-bit signed integers. The least significant half of the 64-bit product is written to the first MCU register. The most significant half is discarded, but if any bit of that half is different from the most significant bit of the least significant half, then the V flag is set; otherwise the V flag is unset.

Syntax

MPY32V <MCU-register-1><MCU-register-2>

where

    <MCU-register-1> ::= <MCU-register>
    <MCU-register-2> ::= <MCU-register>

Notes

1. *MCU-register-1* states the value to be placed in the MCUR field of the instruction, specifying the register which when the instruction is entered will hold the first operand and when the instruction has been completed will hold the least significant half of the product
2. *MCU-register-2* states the value to be placed in the MOD field of the instruction, specifying the register containing the second operand

Possible run-time program errors

None

Example

```
MPY32V  M12  M10          ! FORM THE PRODUCT OF THE CONTENTS
                          ! OF M12 AND M10, AND WRITE THE LEAST
                          ! SIGNIFICANT HALF BACK TO M12.
                          !
                          ! IF THE MOST SIGNIFICANT HALF HAS ANY
                          ! BIT THAT DIFFERS FROM THE MOST SIGNIFICANT
                          ! BIT OF THE LEAST SIGNIFICANT HALF,
                          ! SET THE V FLAG, OTHERWISE UNSET
                          ! THE V FLAG
```

MPY64 MPY64

Function

MPY64 performs a multiplication of the contents of two MCU registers, regarded as 32-bit signed integers. The least significant half of the 64-bit product is written to the first MCU register. The most significant half is written to the register whose number is one less than that of the first MCU register.

Syntax

MPY64 <MCU-register-1><MCU-register-2>

where

<MCU-register-1> ::= M1 | M2 | M3 | M4 | M5 | M6 | M7 | M8 | M9 | M10 | M11 | M12 | M13
<MCU-register-2> ::= <MCU-register>

Notes

1 *MCU-register-1* states the value to be placed in the MCUR field of the instruction, specifying the register which when the instruction is entered will hold the first operand and when the instruction has been completed will hold the least significant half of the product

2 *MCU-register-2* states the value to be placed in the MOD field of the instruction, specifying the register containing the second operand

3 If *MCU-register-1* is M3 (for example) then the 64-bit result is written to M2 and M3, with M2 holding the most significant half

4 M0 cannot be specified as the first operand

Possible run-time program errors

None

Example

```
MPY64  M5  M7              ! FORM THE PRODUCT OF THE CONTENTS
                           ! OF M5 AND M7, PUTTING THE DOUBLE-
                           ! LENGTH RESULT INTO M4 AND M5, WITH THE
                           ! MOST SIGNIFGICANT HALF IN M4
```

MPYU32 MPYU32

Function

MPYU32 performs a multiplication of the contents of two MCU registers, regarded as 32-bit unsigned integers. The least significant half of the 64-bit product is written to the first MCU register. The most significant half is discarded.

Syntax

MPYU32 <MCU-register-1><MCU-register-2>

where

    <MCU-register-1> ::= <MCU-register>
    <MCU-register-2> ::= <MCU-register>

Notes

1. *MCU-register-1* states the value to be placed in the MCUR field of the instruction, specifying the register which when the instruction is entered will hold the first operand and when the instruction has been completed will hold the least significant half of the product
2. *MCU-register-2* states the value to be placed in the MOD field of the instruction, specifying the register containing the second operand

Possible run-time program errors

None

Example

```
MPYU32  M4  M1              ! FORM THE PRODUCT OF THE CONTENTS
                            ! OF M4 AND M1 (REGARDED AS UNSIGNED
                            ! INTEGERS), AND WRITE THE LEAST
                            ! SIGNIFICANT HALF INTO M4
                            !
                            ! THE MOST SIGNIFICANT HALF OF THE
                            ! PRODUCT IS DISREGARDED
```

MPYU32V MPYU32V

Function

MPYU32V performs a multiplication of the contents of two MCU registers, regarded as 32-bit unsigned integers. The least significant half of the 64-bit product is written to the first MCU register. The most significant half is discarded, but if any bit of it is set then the V flag is set; otherwise the V flag is unset.

Syntax

MPYU32V <MCU-register-1><MCU-register-2>

where

<MCU-register-1> ::= <MCU-register>
<MCU-register-2> ::= <MCU-register>

Notes

1 *MCU-register-1* states the value to be placed in the MCUR field of the instruction, specifying the register which when the instruction is entered will hold the first operand and when the instruction has been completed will hold the least significant half of the product

2 *MCU-register-2* states the value to be placed in the MOD field of the instruction, specifying the register containing the second operand

Possible run-time program errors

None

Example

```
MPYU32  M0  M1          ! FORM THE PRODUCT OF THE CONTENTS
                        ! OF M0 AND M1 (REGARDED AS UNSIGNED
                        ! INTEGERS), AND WRITE THE LEAST
                        ! SIGNIFICANT HALF INTO M0.
                        !
                        ! THE V FLAG IS SET EQUAL TO
                        ! THE VALUE OF THE 'OR' FUNCTION OF
                        ! THE BITS IN THE MOST SIGNIFICANT
                        ! HALF OF THE PRODUCT.
```

MPYU64 MPYU64

Function

MPYU64 performs a multiplication of the contents of two MCU registers, regarded as 32-bit unsigned integers. The least significant half of the 64-bit product is written to the first MCU register. The most significant half is written to the register whose number is one less than that of the first operand.

Syntax

MPYU64 <MCU-register-1><MCU-register-2>

where

```
<MCU-register-1> ::= M1 | M2 | M3 | M4 | M5 | M6 | M7 | M8 | M9 | M10 | M11 | M12 |
                     M13
<MCU-register-2> ::= <MCU-register>
```

Notes

1 *MCU-register-1* states the value to be placed in the MCUR field of the instruction, specifying the register which when the instruction is entered will hold the first operand and when the instruction has been completed will hold the least significant half of the product

2 *MCU-register-2* states the value to be placed in the MOD field of the instruction, specifying the register containing the second operand

3 If *MCU-register-1* is M3 (for example) then the 64-bit result is written to M2 and M3, with M2 holding the most significant half

4 M0 cannot be specified as the first operand

Possible run-time program errors

None

Example

```
MPYU64  M2  M1          ! FORM THE PRODUCT OF THE CONTENTS
                        ! OF M2 AND M1, PUTTING THE DOUBLE-
                        ! LENGTH RESULT INTO M2 AND M1 AS
                        ! UNSIGNED NUMBERS, WITH THE MOST
                        ! SIGNIFICANT HALF IN M1
```

NAND

NAND

Function

NAND creates the logical NAND of two MCU registers, putting the result in one of them. The bits from either or both registers can be inverted before doing the logical NAND.

Syntax

NAND <(inverted) MCU-register-1><(inverted) MCU-register-2>

where

<(inverted) MCU-register-1> ::= <(inverted) MCU-register>
<(inverted) MCU-register-2> ::= <(inverted) MCU-register>
<(inverted) MCU-register> ::= <MCU-register> | <inverted MCU-register>

Notes

1 MCU-register-1 specifies the value in the MCUR field of the instruction, which specifies both the register containing the first operand and the register to contain the result

2 MCU-register-2 specifies the value in the MOD field of the instruction which specifies the register containing the second operand

Possible run-time program errors

None.

Example

```
NAND  M1N  M4N              ! M1 = (NOT M1) NAND (NOT M4)
```

NANDH NANDHN

Function

NAND creates the logical NAND of an MCU register and a literal, putting the result in the register. The bits of the register can be inverted before doing the logical NAND.

NANDHN does exactly as NANDH but uses the inverse of the literal.

Syntax

NANDH <(inverted) MCU-register><literal_16>
NANDHN <(inverted) MCU-register><literal_16>

(for further details of:

<(inverted) MCU-register>	see NAND instruction
<literal_16>	see section 6.1.7 for details

Notes

1 *MCU-register* specifies the value in the MCUR field of the instruction which gives both the register containing the first operand and the register to contain the result

2 *literal_16* specifies the value in the LITERAL field of the instruction. To form the second operand, the bit pattern representing the value is extended to 32 bits with leading zeros (see section 6.1.7 for details). In the case of NANDHN this 32-bit literal is inverted before use.

Possible run-time program errors

None

Example

```
NANDHN  M1  "F"          ! M1 = M1 NAND #FFFFDFB9
```

NEQ NEQ

Function

NEQ creates the logical non-equivalence of two MCU registers, putting the result in one of them.

Syntax

NEQ <MCU-register-1><MCU-register-2>

where

```
<MCU-register-1> ::= <MCU-register>
<MCU-register-2> ::= <MCU-register>
```

Notes

1 *MCU-register-1* specifies the value in the MCUR field of the instruction, which specifies both the register containing the first operand and the register to contain the result

2 *MCU-register-2* specifies the value in the MOD field of the instruction which specifies the register containing the second operand

Possible run-time program errors

None.

Example

```
NEQ  M0  M2                 ! SET EACH BIT OF M0 TO THE LOGICAL
                            ! NON_EQUIVALENCE OF THE CORRESPONDING
                            ! BITS OF M0 WITH M2.
```

NEQH NEQH

Function

NEQH creates the logical non-equivalence of an MCU register and a literal putting the result in the register.

Syntax

NEQH <MCU-register><literal_16>

where <literal_16> is defined in section 6.1.7.

Notes

1 *MCU-register* specifies the value in the MCUR field of the instruction which gives both the register containing the first operand and the register to contain the result

2 *literal_16* specifies the value in the LITERAL field of the instruction. To form the second operand, the bit pattern representing the value is expanded to 32 bits with leading zeros (see section 6.1.7 for details)

Possible run-time program errors

None

Example

```
NEQH  M1  16                    ! SET EACH BIT OF M1 TO THE LOGICAL
                                ! NON EQUIVALENCE OF THE CORRESPONDING
                                ! BITS OF M1 WITH #00000010.
```

NOR NOR

Function

NOR creates the logical NOR of two MCU registers, putting the result in one of them. The bits from either or both registers can be inverted before doing the logical NOR.

Syntax

NOR <(inverted) MCU-register-1><(inverted) MCU-register-2>

where

<(inverted) MCU-register-1> ::= <(inverted) MCU-register>
<(inverted) MCU-register-2> ::= <(inverted) MCU-register>
<(inverted) MCU-register> ::= <MCU-register> | <inverted MCU-register>

Notes

1 *MCU-register-1* specifies the value in the MCUR field of the instruction, which specifies both the register containing the first operand and the register to contain the result

2 *MCU-register-2* specifies the value in the MOD field of the instruction, which specifies the register containing the second operand

Possible run-time program errors

None.

Example

```
NOR  M3  M6N                 ! SET EACH BIT OF M3 TO THE LOGICAL
                             ! NOT_OR OF THE CORRESPONDING
                             ! BITS OF M3 WITH INVERTED M6
```

NORH NORHN

Function

NORH creates the logical NOR of an MCU register and a literal, putting the result in the register. The bits of the register can be inverted before doing the logical NOR.

NORHN does exactly as NORH but uses the inverse of the literal.

Syntax

NORH <(inverted) MCU-register><literal_16>
NORHN <(inverted) MCU-register><literal_16>

where

<(inverted) MCU-register>	::= <MCU-register> | <inverted MCU-register>
<literal_16>	see section 6.1.7 for details

Notes

1 *MCU-register* specifies the value in the MCUR field of the instruction, which gives both the register containing the first operand and the register to contain the result

2 *literal_16* specifies the value in the LITERAL field of the instruction. To form the second operand, the bit pattern representing the value is extended to 32 bits with leading zeros (see section 6.1.7 for details). In the case of NORHN, this 32-bit literal is inverted before use

Possible run-time program errors

None

Example

```
NORH  M2N  #FE              ! M2 = (NOT M2) NOR #000000FE
```

NULL NULL

Function

NULL has no effect.

Syntax

NULL

Possible run-time program errors

None.

OR

Function

OR creates the logical OR of two MCU registers, putting the result in one of them. The bits from either or both registers can be inverted before doing the logical OR.

Syntax

OR <(inverted) MCU-register-1><(inverted) MCU-register-2>

where

<(inverted) MCU-register-1> ::= <(inverted) MCU-register>
<(inverted) MCU-register-2> ::= <(inverted) MCU-register>
<(inverted) MCU-register> ::= <MCU-register> | <inverted MCU-register>

Notes

1 *MCU-register-1* specifies the value in the MCUR field of the instruction, which specifies both the register containing the first operand and the register to contain the result

2 *MCU-register-2* specifies the value in the MOD field of the instruction which specifies the register containing the second operand

Possible run-time program errors

None.

Example

```
OR  M0N  M1                    ! SET EACH BIT OF M0 TO THE LOGICAL
                               ! OR OF M1 WITH INVERTED M0.
```

ORH ORHN

Function

ORH creates the logical OR of an MCU register and a literal, putting the result in the register. The bits of the register can be inverted before doing the logical OR.

ORHN does exactly as ORH but uses the inverse of the literal

Syntax

ORH <(inverted) MCU-register><literal_16>
ORHN <(inverted) MCU-register><literal_16>

where

<(inverted) MCU-register>	::= <MCU-register> | <inverted MCU-register>
<literal_16>	see section 6.1.7 for details

Notes

1 *MCU-register* specifies the value in the MCUR field of the instruction which specifies both the register containing the first operand and the register to hold the result

2 *literal_16* specifies the value in the LITERAL field of the instruction. To form the second operand, the bit pattern representing the value is extended to 32 bits with leading zeros (see section 6.1.7 for details). In the case of ORHN this 32-bit literal is inverted before use

Possible run-time program errors

None

Example

```
ORHN  M1  -1                    ! M1=M1 OR  #FFFF0000
```

PAUSE

Function

PAUSE is a pseudo instruction that causes the DAP program to halt temporarily, with the option of its being restarted; it is one of several instructions (PAUSE, STOP and TRACE) that the assembler encodes as particular cases of the CALL instruction. The action taken on PAUSE by the run-time-support system is specified by you prior to running the program, and can be depend on the *number* associated with the PAUSE statement. The options you have available include, in various combinations:

- Abandon the DAP and host programs
- Initiate a diagnostic dump of DAP store contents
- Enter the interactive program state analysis mode
- Restart the DAP program

In the simulation environment, one use of PAUSE is to provide timing information.

For further details see *DAP Series: Program Development*.

Syntax

PAUSE <pause number>?

where <pause number> ::= <numval>

Notes

1 *pause number* must be in the range 1 to 262, 143. The action taken can depend on the value of this number

Possible run-time program errors

None.

QA QAN

Function

QA copies the A plane into the Q plane.

QAN copies the inverse of the A plane to the Q plane.

Syntax

QA
QAN

Possible run-time program errors

None.

QA_CF QAN_CF

Function

QA_CF is a compound instruction. It copies the A plane into the Q plane, then sets every bit of the C plane to zero.

QAN_CF copies the inverse of the A plane into the Q plane and then sets every bit of the C plane to zero.

Syntax

QA_CF
QAN_CF

Possible run-time program errors

None.

QB QBN

Function

QB sets every bit of the Q plane to a specified bit of an MCU register or the edge register

QBN sets every bit of the Q plane to the inverse of a specified bit of an MCU register or the edge register

Syntax

QB <MCU-or-edge-register>.<bit number><modifier>?<step>?
QBN <MCU-or-edge-register>.<bit number><modifier>?<step>?

where <MCU-or-edge-register>, <bit number>, <modifier>, and <step> together form a register bit address (see section 7.3.4 for details)

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective INT field defines the bit number of the MCU register or the edge register, and the effective ADDR field is discarded.

Notes

See notes for the AB instruction

Possible run-time program errors

None

Example

```
QB  M4.0  (M1)  (+)          ! SET EACH BIT OF THE Q PLANE TO
                             ! BIT i OF M4, WHERE i IS
                             ! MODULO 32 OF ((INT FIELD OF M1) + DO
                             ! LOOP ITERATION NUMBER)
```

QC QCN

Function

QC copies the C plane into the Q plane.

QCN copies the inverse of the C plane into the Q plane.

Syntax

QC
QCN

Possible run-time program errors

None.

QC_CF QC_CF

Function

QC_CF is a compound instruction. It copies the C plane into the Q plane, then sets every bit of the C plane to zero.

Syntax

QC_CF

Possible run-time program errors

None.

QEBS QEBSN

Function

QEBS creates the logical equivalence of each bit of a given store plane with a specified bit of an MCU register or the edge register. The result is put in the Q plane.

QEBSN is as QEBS, but uses the inverse of the store plane.

Syntax

QEBS <MCU-or-edge-register>.<bit number><plane><modifier>?<step>?
QEBSN <MCU-or-edge-register>.<bit number><plane><modifier>?<step>?

where

> <MCU-or-edge-register>, <bit number>, <modifier>, and <step> together form a register bit address (see section 7.3.4 for details)
>
> <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details)

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field defines the bit number of the MCU register or the edge register.

Notes

See notes for the AEBS instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
QEBS M4.0  SPLANE  (M3)  (+)        ! SET EACH BIT OF THE Q PLANE
                                    ! TO THE LOGICAL EQUIVALENCE OF
                                    ! THE CORRESPONDING BIT OF STORE
                                    ! PLANE SPLANE ((M3) + i) WITH BIT
                                    ! ((INT FIELD OF M3 + i) MOLULO 32) OF M4,
                                    ! WHERE i IS THE DO LOOP
                                    ! ITERATION NUMBER.
```

QF

Function

QF sets every bit of the Q plane to zero.

Syntax

QF

Possible run-time program errors

None.

QF_AF QF_AF

Function

QF_AF is a compound instruction. It sets every bit of the Q and A planes to zero.

Syntax

QF_AF

Possible run-time program errors

None.

QF_CF QF_CF

Function

QF_CF is a compound instruction. It sets every bit of the C and Q planes to zero.

Syntax

QF_CF

Possible run-time program errors

None.

QPCA QPCAN

Function

QPCA adds corresponding bits of the C and A planes, putting the sums in the Q plane.

QPCAN is as QPCA but uses the inverse of the A plane.

The carry bits are discarded.

Syntax

QPCA
QPCAN

Possible run-time program errors

None.

QPCQ QPCQ

Function

QPCQ adds corresponding bits of the C and Q planes, putting the sums in the Q plane. The carry bits are discarded.

Syntax

QPCQ

Possible run-time program errors

None.

QPCQA QPCQAN

Function

QPCQA adds corresponding bits of the C, Q and A planes, putting the sums in the Q plane.

QPCQAN is as QPCQA but uses the inverse of the A plane.

The carry bits are discarded.

Syntax

QPCQA
QPCQAN

Possible run-time program errors

None.

QPCQR QPCQRN

Function

QPCQR adds corresponding bits of the C, Q and R planes, putting the sums in the Q plane.

QPCQRN is as QPCQR but uses the inverse of the R plane.

The carry bits are discarded.

Syntax

QPCQR <MCU-or-edge-register>
QPCQRN <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction which specifies the register that is used to form the R plane (see section 6.2 for details)

Possible run-time program errors

None.

QPCQRO QPCQRNO

Function

QPCQRO adds corresponding bits of the C, Q and orthogonal R planes, putting the sums in the Q plane.

QPCQRNO is as QPCQRO but uses the inverse of the orthogonal R plane.

The carry bits are discarded.

Syntax

QPCQRO <MCU-or-edge-register>
QPCQRNO <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction which specifies the register that is used to form the orthogonal R plane (see section 6.2 for details)

Possible run-time program errors

None.

QPCQS QPCQSN

QPCQS QPCQSN

Function

QPCQS adds corresponding bits of the C and Q planes and a given store plane putting the sums in the Q plane.

QPCQSN is as QPCQS but uses the inverse of the store plane.

The carry bits are discarded.

Syntax

QPCQS <plane><modifier>?<step>?
QPCQSN <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details)

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the QS instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
QPCQS  SPLANE  (M2)          ! SET EACH BIT OF THE Q PLANE TO
                             ! THE SUM OF THAT BIT AND
                             ! THE CORRESPONDING BITS OF THE C PLANE
                             ! AND STORE PLANE SPLANE (M2)
```

QPCQT QPCQT

Function

QPCQT adds corresponding bits of the C and Q planes and a notional plane consisting of all ones, putting the sums in the Q plane. The carry bits are discarded.

Syntax

QPCQT

Possible run-time program errors

None.

QPCR QPCRN

Function

QPCR adds corresponding bits of the C and R planes, putting the sums in the Q plane.

QPCRN is as QPCR but uses the inverse of the R plane.

The carry bits are discarded.

Syntax

QPCR <MCU-or-edge-register>
QPCRN <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction which specifies the register that is used to form the R plane (see section 6.2 for details)

Possible run-time program errors

None.

QPCRO
QPCRNO

Function

QPCRO adds corresponding bits of the C and orthogonal R planes, putting the sums in the Q plane.

QPCRNO is as QPCRO but uses the inverse of the orthogonal R plane.

The carry bits are discarded.

Syntax

QPCRO <MCU-or-edge-register>
QPCRNO <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value of the MCUR field of the instruction which specifies the register that is used to form the orthogonal R plane (see section 6.2 for details)

Possible run-time program errors

None.

QPCS QPCSN

Function

QPCS adds corresponding bits of the C plane and a given store plane, putting the sums in the Q plane.

QPCSN is as QPCS but uses the inverse of the store plane.

The carry bits are discarded.

Syntax

QPCS <plane><modifier>?<step>?
QPCSN <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details)

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the QS instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
QPCS  S1  (-)                 ! SET EACH BIT OF THE Q PLANE TO THE SUM
                              ! OF THE CORRESPONDING BITS OF THE C PLANE
                              ! AND STORE PLANE S1 - i, WHERE i IS
                              ! THE DO LOOP ITERATION NUMBER.
```

QPQA QPQAN

Function

QPQA adds corresponding bits of the Q and A planes, putting the sums in the Q plane.

QPQAN is as QPQA but uses the inverse of the A plane.

The carry bits are discarded.

Syntax

```
QPQA
QPQAN
```

Possible run-time program errors

None.

QPQR QPQRN

Function

QPQR adds corresponding bits of the Q and R planes, putting the sums in the Q plane.

QPQRN is as QPQR but uses the inverse of the R plane.

The carry bits are discarded.

Syntax

QPQR <MCU-or-edge-register>
QPQRN <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction which specifies the register that is used to form the R plane (see section 6.2 for details)

Possible run-time program errors

None.

QPQRO QPQRNO

Function

QPQRO adds corresponding bits of the Q and orthogonal R planes, putting the sums in the Q plane.

QPQRNO is as QPQRO but uses the inverse of the orthogonal R plane.

The carry bits are discarded.

Syntax

QPQRO <MCU-or-edge-register>
QPQRNO <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction which specifies the register that is used to form the orthogonal R plane (see section 6.2 for details)

Possible run-time program errors

None.

QPQS QPQSN

QPQS QPQSN

Function

QPQS adds corresponding bits of the Q plane and a given store plane, putting the sums in the Q plane.

QPQSN is as QPQS but uses the inverse of the store plane.

The carry bits are discarded.

Syntax

QPQS <plane><modifier>?<step>?
QPQSN <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details)

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the QS instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
QPQS  49  (M6)              ! SET EACH BIT OF THE Q PLANE TO THE SUM OF
                            ! THAT BIT AND THE CORRESPONDING BIT OF
                            ! STORE PLANE 49 + (ADDR FIELD OF M6)
```

QQ QQ

Function

QQ shifts the Q plane a specified number of places.

Syntax

QQ <direction>?<geometry>?<count>?<modifier><step>?
QQ <nesw><geometry><count>?<step>?

where

> <direction> ::= <nesw> | R0 | R1 | R2 | R3
> <geometry> ::= P | C | PC | CP
> <count> ::= <numval>
> <nesw> ::= N | E | S | W

Addressing Mode

Mode D addressing is used (see section 7.1.4 for details). The result specifies the effective direction, geometry and count. If the effective count value is zero, the instruction has no effect.

Notes

See notes 1 to 4 for AMQ(shifting).

Possible run-time program errors

None

Example

```
QQ  W  P  1  (M2)          ! SHIFT THE Q PLANE
                           ! 1 + (INT FIELD OF M2) MODULO ES
                           ! PLACES TO THE WEST, USING PLANE GEOMETRY.
```

QQN QQN

Function

QQN inverts every bit of the Q plane.

Syntax

QQN

Possible run-time program errors

None.

QR QRN

Function

QR copies the R plane into the Q plane.

QRN copies the inverse of the R plane into the Q plane.

Syntax

QR <MCU-or-edge-register>
QRN <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction which specifies the register that is used to form the R plane (see section 6.2)

Possible run-time program errors

None.

Example

```
QR  M3                 ! EACH ROW OF THE Q PLANE IS SET EQUAL TO THE
                       ! CONTENTS OF M3; M3 IS FIRST EXTENDED TO THE
                       ! LEFT WITH ZEROS (IF NECESSARY) TO GIVE AN
                       ! ES-SIZED VALUE.
```

QRO QRNO

Function

QRO copies the orthogonal R plane into the Q plane.

QRNO copies the inverse of the orthogonal R plane into the Q plane.

Syntax

QRO <MCU-or-edge-register>
QRNO <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction which specifies the register that is used to form the orthogonal R plane (see section 6.2 for details)

Possible run-time program errors

None.

Example

```
QRO  M5                 ! EACH COLUMN OF THE Q PLANE IS SET EQUAL TO THE
                        ! CONTENTS OF M5. M5 IS FIRST EXTENDED TO THE
                        ! LEFT WITH ZEROS (IF NECESSARY) TO GIVE AN
                        ! ES-SIZED VALUE.
```

QS QSN

Function

QS copies a given store plane into the Q plane.

QSN copies the inverse of a given store plane into the Q plane.

Syntax

QS <plane><modifier>?<step>?
QSN <plane><modifier>?<step>?

where <plane>,<modifier>, and <step> together form a store plane address, as described in section 7.3.3

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

1 *plane* specifies the value in the ADDR field of the instruction, which is used to construct the effective ADDR value

2 *modifier* specifies the value in the MOD field of the instruction which specifies the modifier register to be used, if any

3 *step* specifies the value in the INCREMENT/DECREMENT field of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
QS  SPLANE  (M2)  (+)        ! EACH BIT OF THE Q PLANE IS SET TO
                             ! THE CORRESPONDING BIT OF STORE
                             ! PLANE SPLANE (M2) + i, WHERE i IS THE
                             ! DO LOOP ITERATION NUMBER
```

QS_AS QSN_ASN

Function

QS_AS is a compound instruction. It copies a given store plane into the Q and A planes.

QSN_ASN copies the inverse of a given store plane into the Q and A planes.

Syntax

QS_AS <plane><modifier>?<step>?
QSN_ASN <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details)

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the QS instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
QS_AS  20  (M2)              ! EACH BIT OF THE Q AND A PLANES IS SET
                             ! TO THE CORRESPONDING BIT OF STORE PLANE
                             ! 20 + (ADDR FIELD OF M2).
```

QS_CF QSN_CF

Function

QS_CF is a compound instruction. It copies a given store plane into the Q plane, then sets every bit of the C plane to zero.

QSN_CF copies the inverse of a given store plane into the Q plane, then sets every bit of the C plane to zero.

Syntax

QS_CF <plane><modifier>?<step>?
QSN_CF <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details)

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the QS instruction.

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
QS_CF SPLANE + 2 (M3) (-)       ! EACH BIT OF THE C PLANE IS
                                ! SET TO ZERO, AND EACH BIT OF
                                ! THE Q PLANE IS SET TO THE
                                ! CORRESPONDING BIT OF STORE
                                ! PLANE SPLANE (M3) + 2 - i, WHERE i IS
                                ! THE DO LOOP ITERATION NUMBER.
```

QT QT

Function

QT sets every bit of the Q plane to one.

Syntax

QT

Possible run-time program errors

None.

QT_AT

Function

QT_AT is a compound instruction. It sets every bit of both the Q plane and A plane to one.

Syntax

QT_AT

Possible run-time program errors

None.

QT_CF QT_CF

Function

QT_CF is a compound instruction. It sets every bit of the Q plane to one and every bit of the C plane to zero.

Syntax

QT_CF

Possible run-time program errors

None.

QVCQ QVCQ

Function

QVCQ adds together corresponding rows or columns of the C and Q planes treating each pair of rows or columns as *ES*-sized unsigned integers. The results of the additions are placed in the Q plane; the C plane is not altered by this instruction.

Syntax

QVCQ <direction>?<geometry>?<count>?<modifier><step>?
QVCQ <nesw><geometry><count>?<step>?

where

<direction> :: = <nesw> | R0 | R1 | R2 | R3
<geometry> :: = P | C | PC | CP
<count> :: = <numval>
<nesw> :: = N | E | S | W

Addressing Mode

Mode D addressing is used (see section 7.1.4 for details). The result specifies the effective direction, geometry and count.

Notes

See notes for the CQVCQ instruction.

Possible run-time program errors

None

Example

```
QVCQ  W  P  15
```

In this example, on DAP 500 the Q and C planes are added as 32 rows of integers. Notional bit position 32 is a defined carry point because of plane geometry. A count of 15 guarantees 16 result sum and carry-out bits in positions 16 to 31 of each row. The sum bits go into the Q plane; the C plane is unchanged.

RAC RAC

Function

RAC is a pseudo instruction that loads an instruction address into an MCU register. The instruction address is created in the literals area and accessed in the same way as other literals. Its format is identical to the link value created by a JSL instruction (see section 7.2.4 for details), but note that the most significant bits of the value are undefined. The instruction referenced must be in the same code section as the RAC instruction.

Syntax

RAC <MCU-register><code label name><label offset>?

where <code label name> and <label offset> together form a within-section instruction address as described in section 7.3.5.

Notes

1 An error will occur during consolidation if the instruction address is not in the same code section as the RAC instruction

Possible run-time program errors

None.

Example

```
RAC  M12  LABEL2              ! LOAD ADDRESS OF LABEL2 INTO M12
```

RACE RACE

Function

RACE is a pseudo instruction that loads an instruction address into an MCU register. The instruction address is created in the literals area and accessed in the same way as other literals. Its format is identical to the link value created by a JSL instruction (see section 7.2.4 for details), but note that the most significant bits of the value are undefined. The instruction referenced is identified relative to the first instruction of a named entry point in the same or another code section.

Syntax

RACE <MCU-register><code section name><section offset>?
RACE <MCU-register><entry point name><section offset>?

where <code section name> (or <entry point name>) and <section offset> together form an instruction address as described in section 7.3.5.

Possible run-time program errors

None.

Example

```
RACE  M12  SUB1               ! LOAD ADDRESS OF ENTRY POINT SUB1 INTO M12
```

RALITR RALITR

Function

RALITR is a pseudo instruction (see section 6.2 for details) which creates a row-aligned literal value and loads the address of the first row of that value into a specified MCU register.

Syntax

RALITR <MCU-register><value><size>?

where <value> and <size> are as defined below:

Type of value	*Size, in bits*	
	Range of possible sizes	Default size
integer	1 to 64, in steps of 1	32
real	24 to 64, in steps of 8	32
hexadecimal	1 to (64 or *ES*, whichever is the greater) in steps of 1	32
character	8 to 64, in steps of 8	32

The instructions RALITR, RALITW and RLIT create literals for which the same limits to <value>s and <size>s apply.

These <value>s and <size>s are much the same as for items in a data section (see section 4.2.3 for more details), except that the RALITR maximum <size> for character values is 64 bits (in a data section the maximum size is 512 bits, that is 64 characters), and that the RALITR default <size> for character values is 32 bits (in a data section the default in bits is 8 times the number of characters the item is initialised with).

As with data items, integers are created in two's complement form.

Possible run-time program errors

None.

Examples

```
RALITR  M5  -1(24)          ! CREATE A LITERAL HAVING THE VALUE
                            ! -1 OCCUPYING THE LEAST SIGNIFICANT
                            ! 24 BITS OF A ROW, AND LOAD THE ADDRESS
                            ! OF THAT ROW INTO M5
```

Examples – continued

```
RALITR  M2  "ABC"         ! CREATE A LITERAL HAVING THE VALUE "ABC"
                          ! AND EXTENDED BY ONE SPACE TO THE LEFT
                          ! TO OCCUPY THE LEAST SIGNIFICANT 32 BITS
                          ! OF A ROW, AND LOAD THE ADDRESS OF THE
                          ! ROW CONTAINING THAT LITERAL INTO M2.
```

RALITW RALITW

Function

RALITW is a pseudo instruction (see section 6.2 for details) which creates a word-aligned literal value and loads the address of the first word of that value into a specified MCU register.

Syntax

RALITW <MCU-register><value><size>?

where <value> and <size> are as defined below:

Type of value	*Size, in bits*	
	Range of possible sizes	Default size
integer	1 to 64, in steps of 1	32
real	24 to 64, in steps of 8	32
hexadecimal	1 to (64 or *ES*, whichever is the greater) in steps of 1	32
character	8 to 64, in steps of 8	32

The instructions RALITR, RALITW and RLIT create literals for which the same limits to <value>s and <size>s apply.

These <value>s and <size>s are much the same as for items in a data section (see section 4.2.3 for more details), except that the RALITR maximum <size> for character values is 64 bits (in a data section the maximum size is 512 bits, that is 64 characters), and that the RALITR default <size> for character values is 32 bits (in a data section the default in bits is 8 times the number of characters the item is initialised with).

As with data items, integers are created in two's complement form.

Notes

1 On DAP 500, words and rows are equivalent, so RALITW has the same effect as RALITR

Possible run-time program errors

None.

Examples

```
RALITW  M5  -1(24)      ! CREATE A LITERAL HAVING THE VALUE
                        ! -1 OCCUPYING THE LEAST SIGNIFICANT
                        ! 24 BITS OF A WORD, AND LOAD THE ADDRESS
                        ! OF THAT WORD INTO M5

RALITW  M3  2.5         ! CREATE A LITERAL HAVING THE REAL VALUE
                        ! 2.5 AND OCCUPYING ONE 32-BIT WORD, AND
                        ! LOAD THE ADDRESS OF THAT WORD INTO M3.
```

RANO RANO

Function

RANO sets a specified MCU register or the edge register to the logical AND of all *ES* columns of the inverse of the A plane.

Syntax

RANO <MCU-or-edge-register>

Notes

1 *MCU-or edge-register* specifies the value in the MCUR field of the instruction which specifies the register to contain the result

2 The AND function produces an *ES*-size value, and if ME is specified then this entire value is written into ME. If an MCU register is specified, then the least significant 32 bits at the *ES*-sized value are written into the register and any other bits discarded

Possible run-time program errors

None.

Example

```
RANO  ME                 ! SET EACH BIT OF THE EDGE REGISTER TO THE
                         ! LOGICAL AND OF THE INVERSE OF ALL THE BITS
                         ! IN THE CORRESPONDING A PLANE ROW
```

RAPL

Function

RAPL is a pseudo instruction (see section 6.2 for details) which loads the plane part of a specified data address into the ADDR field of a specified MCU register. Remaining bits of the register are set to zero.

If possible, an RAX instruction is generated; otherwise a literal is created to hold the address, and an instruction is generated to load the address from the literals area.

Syntax

RAPL <MCU-register><data address>

where <data address> is defined in section 7.3.3.

Notes

1 Only the plane part of the address is used, but the address need not be plane or row aligned

2 The value loaded into the MCU register is discussed in section 7.2.1

Possible run-time program errors

None.

Example

```
DATA SECT1
   FRED: 3 * PLANE
   TOM: 3.142
END
.
.
.
RAPL M3 TOM + 2            ! LOAD THE VALUE n + 5 INTO THE ADDR FIELD
                           ! OF M3. n IS THE PLANE ADDRESS OF DATA
                           ! SECTION SECT1. TOM HAS A PLANE
                           ! DISPLACEMENT OF 3 WITHIN SECT1, AND
                           ! ANOTHER DISPLACEMENT OF 2 PLANES IS
                           ! SPECIFIED IN THE INSTRUCTION
```

RAR RAR

Function

RAR is a pseudo instruction (see section 6.2 for details) which loads both the plane and row parts of a specified address into a specified MCU register. The remaining bits of the register are set to zero.

If possible, an RAX instruction is generated; otherwise a literal is created to hold the address, and an instruction is generated to load the address from the literals area.

Syntax

RAR <MCU-register><data address>

where <data address> is defined in section 7.3.3.

Notes

1 Only the plane and row parts of the address are used, but the address need not be word aligned

2 The value loaded into the MCU register is discussed in section 7.2.1

Possible run-time program errors

None.

Example

```
DATA DATASEC
   FRED : 200*PLANE
   JOE : PLANE
   TOM : PLANE
END
.
.
.
RAR M4 JOE .37                 ! LOAD THE ADDRESS OF ROW 37 OF JOE INTO M4.
                               ! ON DAP 500 THIS ADDRESS HAS AN OFFSET OF
                               ! ONE PLANE AND FIVE ROWS FROM JOE, WHICH
                               ! IN THIS CASE IS EQUIVALENT TO ROW 5 OF
                               ! TOM. ON DAP 600 ROW 37 OF PLANE JOE IS
                               ! ADDRESSED
```

RASC RASC

Function

RASC is a pseudo instruction (see section 6.2 for details) which loads the address of the start of the data section, or data part of a mixed section, associated with a specified data address into a given MCU register.

A literal is created to hold the address, and an instruction is generated to load the address from the literals area.

Syntax

RASC <MCU-register><data address>

where <data address> is defined in section 7.3.3.

Notes

1 The value loaded into the MCU register is discussed in section 7.2.1

Possible run-time program errors

None.

Example

```
RASC M6 VAR1                ! LOAD INTO M6 THE START ADDRESS OF THE
                            ! DATA SECTION IN WHICH VAR1 IS DECLARED.
```

RAW RAW

Function

RAW loads the address of a word into a specified MCU register. The effective ADDR, effective INT and (where applicable) effective WORD fields are loaded into the corresponding fields of an MCU register.

Syntax

RAW <MCU-register><word><modifier>?

where <word> and <modifier> together form a store word address (see section 7.3.3 for details).

Addressing Mode

Mode C addressing is used (see section 7.1.3 for details). The effective ADDR, INT and (where applicable) WORD fields together define the store word address.

Notes

1 *MCU-register* specifies the value in the MCUR field of the instruction which specifies the register into which the store address is to be loaded

2 *word* specifies the values in the ADDR, INT and WORD fields of the instruction, which are used to construct the effective ADDR, INT and WORD values

3 *modifier* specifies the value in the MOD field of the instruction which specifies the modifier register to be used, if any. The format of the modifier is given in section 7.2.1

4 The value loaded into the MCU register is discussed in section 7.2.1

Possible run-time program errors

None

Example

```
RAW  M4  12..6             ! LOAD INTO M4 THE ADDRESS, PLANE 12 WORD 6

RAW  M4  12.6              ! LOAD INTO M4 THE ADDRESS, PLANE 12 ROW 6.
                           !
                           ! ON DAP 500 WORDS AND ROWS ARE EQUIVALENT,
                           ! SO THIS AND THE ABOVE EXAMPLE HAVE THE
                           ! SAME EFFECT

RAW  M4  12..6  (M5)       ! AS THE FIRST EXAMPLE ABOVE, EXCEPT THAT
                           ! THE ADDRESS IS MODIFIED BY THE CONTENTS
                           ! OF M5, AS DESCRIBED IN SECTION 7.1.3
```

RAWD RAWD

Function

RAWD is a pseudo instruction (see section 6.2 for details) which loads the plane, row and word parts of the data address associated with a given word into a specified MCU register. The remaining bits of the register are set to zero.

If possible, an RAW instruction is generated; otherwise a literal is created to hold the address, and an instruction is generated to load the address from the literals area.

Syntax

RAWD <MCU-register><word>

where <word> is defined in section 7.3.3.

Notes

1 The value loaded into the MCU register is discussed in section 7.2.1

Possible run-time program errors

None.

Example

```
DATA DATASEC
   FRED : 200*PLANE
   JOE : PLANE
   TOM : 5*PLANE
END
.
.
.
RAWD M4 JOE..131            ! LOAD THE ADDRESS OF WORD 131 OF JOE INTO M4
                            ! ON DAP 600 THIS IS EQUIVALENT TO LOADING
                            ! THE ADDRESS OF WORD 3 OF TOM INTO M4;
                            ! ON DAP 500 IT IS EQUIVALENT TO LOADING
                            ! THE ADDRESS OF WORD 3 OF PLANE (TOM + 3)
                            ! INTO M4
```

RAX RAX

Function

RAX loads the address of a row into a specified MCU register. The effective ADDR, effective INT and effective WORD fields are loaded into the corresponding fields of an MCU register.

Syntax

RAX <MCU-register><row><modifier>?

where <row> and <modifier> together form a store row address (see section 7.3.3 for details).

Addressing mode

Mode B addressing is used (see section 7.1.2 for details). The effective ADDR field and the effective INT field together define the store row address.

Notes

1 *MCU-register* specifies the value in the MCUR field of the instruction which specifies the register into which the store address is to be loaded

2 *row* specifies the values in the ADDR and INT fields of the instruction, which are used to construct the effective ADDR and INT values

3 *modifier* specifies the value in the MOD field of the instruction which specifies the modifier register to be used, if any. The format of the modifier is given in section 7.2.1

4 The value loaded into the MCU register is discussed in section 7.2.1

Possible run-time program errors

None

Examples

```
RAX  M2  14.2  (M6)          ! LOAD ADDR FIELD OF M2 WITH
                             ! 14 + (ADDR FIELD OF M6) + CARRY OUT OF INT FIELD.
                             ! LOAD INT FIELD OF M2 WITH 2 + (INT FIELD OF M6).
RAX  M3  1(M3)               ! ADD 1 PLANE TO THE ADDRESS IN M3.
RAX  M4  0.1  (M4)           ! ADD 1 ROW TO THE ADDRESS IN M4.
```

RDGC RDGC

Function

RDGC is a pseudo instruction (see section 6.2 for details) which loads a specified MCU register with direction, geometry, and count fields. All other bits of the register are set to zero.

An RH instruction is generated.

Syntax

RDGC <MCU-register><nesw><geometry><count>?

where

<nesw>::=N | E | S | W
<geometry>::=P | C | PC | CP
<count>::=<numval>

Notes

1 *MCU-register* specifies the value in the MCUR field of the instruction; the value specifies the register into which the DIRECTION, GEOMETRY, and COUNT values are to be loaded

2 *nesw* specifies the value of DIRECTION

3 *geometry* specifies the value of GEOMETRY

The values are interpreted as follows:

Value	*Geometry*
P	Plane geometry for all shifts
C	Cyclic geometry for all shifts
PC	Plane geometry for north and south shifts, cyclic geometry for east and west shift
CP	Cyclic geometry for north and south shifts, plane geometry for east and west shift

4 *count* specifies the value of COUNT. If *count* is omitted, a zero count is assumed

5 The value loaded into the MCU register is discussed in section 7.2.2

Possible run-time program errors

None.

Example

```
RDGC  M1  N  P  3          ! LOAD M1 WITH A DIRECTION
                           ! OF N, A GEOMETRY OF P, AND
                           ! A COUNT OF 3
```

RF

Function

RF sets every bit in a specified MCU register to zero.

Syntax

RF <MCU-register>

Notes

1 *MCU-register* specifies the value in the MCUR field of the instruction which specifies the register whose bits are to be set to zero

Possible run-time program errors

None.

RH
RHN

Function

RH loads a specified MCU register with a literal.

RHN loads a specified MCU register with the inverse of a literal.

Syntax

```
RH <MCU-register><literal_16>
RHN <MCU-register><literal_16>
```

where <literal_16> is defined in section 6.1.7.

Notes

1 *MCU-register* specifies the value in the MCUR field of the instruction which gives the register to be loaded with the literal

2 *literal_16* specifies the value in the LITERAL field of the instruction. To form the second operand, the bit pattern representing the value is extended to 32 bits with leading zeros (see section 6.1.7 for details). If you are using RH, this bit pattern is then loaded into the register. If you are using RHN, the 32-bit literal is inverted before being loaded into the register

Possible run-time program errors

None

Examples

```
RH  M1  -2(5)            ! M1 = #0000001E
RHN  M2  33              ! M2 = #FFFFFFDE
```

RLIT

Function

RLIT is a pseudo instruction (see section 6.2 for details) which loads a literal value into a specified MCU register or the edge register .

If possible, an RH or RHN instruction is generated; otherwise a literal is created and an instruction is generated to load it from the literals area.

Syntax

RLIT <MCU-or-edge-register><value><size>?

where <value> and <size> are as defined below:

Type of value	*Size, in bits*	
	Range of possible sizes	Default size
integer	1 to 64, in steps of 1	32
real	24 to 64, in steps of 8	32
hexadecimal	1 to (64 or ES, whichever is the greater) in steps of 1	32
character	8 to 64, in steps of 8	32

The instructions RALITR, RALITW and RLIT create literals for which the same limits to <value>s and <size>s apply.

These <value>s and <size>s are much the same as for items in a data section (see section 4.2.3 for more details), except that the RALITR maximum <size> for character values is 64 bits (in a data section the maximum size is 512 bits, that is 64 characters), and that the RALITR default <size> for character values is 32 bits (in a data section the default in bits is 8 times the number of characters the item is initialised with).

As with data items, integers are created in two's complement form.

Possible run-time program errors

None.

Example

```
RLIT  M5  3.247(24)          ! LOAD M5 WITH THE 24-BIT REAL
                             ! VALUE 3.247. THE VALUE WILL HAVE
                             ! BEEN GENERATED AS A LITERAL.
```

RQO RQO

Function

RQO sets a specified MCU register or the edge register to the logical AND of all the columns in the Q plane.

Syntax

RQO <MCU-or-edge-register>

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction which specifies the register to contain the result

2 The AND function produces an *ES*-size value, and if ME is specified then this entire value is written into ME. If an MCU register is specified, then the least significant 32 bits of the *ES*-sized value are written into the regular and any other bits discarded

Possible run-time program errors

None.

Example

```
RQO  ME                       ! SET EACH BIT OF THE EDGE REGISTER TO
                              ! THE LOGICAL AND OF ALL BITS IN
                              ! THE CORRESPONDING Q PLANE ROW
```

RR RRN

Function

RR copies one MCU register to another.

RRN copies the inverse of one MCU register to another.

Syntax

RR <MCU register-1><MCU register-2>
RRN <MCU register-1><MCU register-2>?

where

<MCU register-1> ::= <MCU-register>
<MCU register-2> ::= <MCU-register>

Notes

1 *MCU-register-1* specifies the value in the MCUR field of the instruction, which specifies the register to contain the result

2 *MCU-register-2* specifies the value in the MOD field of the instruction, which specifies the register holding the required value. If *MCU-register-2* is omitted in an RRN instruction, it is assumed to be the same as *MCU-register-1*; that is, the instruction will invert the specified MCU register

Possible run-time program errors

None.

Examples

```
RR  M0  M6              ! LOAD M0 WITH M6
RRN  M2  M3             ! LOAD M2 WITH THE INVERSE OF M3
RRN  M4                 ! LOAD M4 WITH THE INVERSE OF M4
```

RS RSO

Function

RS sets a specified MCU register or the edge register to the logical AND of all the rows in a store plane.

RSO sets a specified MCU register or the edge register to the logical AND of all the columns in a store plane.

Syntax

```
RS <MCU-or-edge-register><plane><modifier>?<step>?
RSO <MCU-or-edge-register><plane><modifier>?<step>?
```

where <plane>, <modifier> and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction which specifies the register to contain the result

2 *plane* specifies the value in the ADDR field of the instruction, which is used to construct the effective ADDR value

3 *modifier* specifies the value in the MOD field of the instruction which specifies the modifier register to be used, if any

4 *step* specifies the value in the INCREMENT/DECREMENT field of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop

5 The AND function produces an *ES*-sized value, and if ME is specified then this entire value is written into ME. If an MCU register is specified, then the least significant 32 bits of the *ES*-sized value are written into the register and any other bits discarded

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
RS  ME  62  (M3)  (-)        ! SET THE EDGE REGISTER TO THE LOGICAL AND OF
                             ! ALL ROWS IN STORE PLANE
                             ! 62 + (ADDR FIELD OF M3) - i,
                             ! WHERE i IS THE DO LOOP ITERATION
                             ! NUMBER
```

RT RT

Function

RT sets every bit of a given MCU register to one.

Syntax

RT <MCU-register>

Notes

1 *MCU-register* specifies the value in the MCUR field of the instruction which specifies the register whose bits are to be set to one

Possible run-time program errors

None.

RW
RWO

RW
RWO

Function

RW copies a given store word into an MCU register.

RWO copies a given store orthogonal word into an MCU register.

Syntax

RW <MCU-register><word><modifier>?<step>?
RWO <MCU-register><word><modifier>?<step>?

where <word>, <modifier> and <step> together form a store word address (see section 7.3.3 for details).

Addressing Mode

Mode C addressing is used (see section 7.1.3 for details). The effective ADDR field, the effective INT field and the effective WORD field together define the store word address.

Notes

1. *MCU-register* specifies the value in the MCUR field of the instruction which specifies the register into which the contents of the addressed word are to be loaded
2. *word* specifies the values in the ADDR, INT and WORD fields of the instruction, which are used to construct the effective ADDR, INT and WORD values
3. The orthogonal word of RWO is obtained by using the effective ADDR and INT fields to select a column of array store, then (where applicable) using the effective WORD address to select a word from that column
4. *modifier* specifies the value in the MOD field of the instruction which specifies the modifier register to be used, if any
5. *step* specifies the value in the INCREMENT/DECREMENT field of the instruction, which specifies whether the word address is to be incremented or decremented if it appears in an APAL DO loop

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
RW  M9  15.23.1          ! COPY INTO REGISTER M9 CONTENTS OF WORD AT
                         ! LOCATION (PLANE 15, ROW 23, WORD 1) IN STORE.

RWO  M12  6.33.0         ! COPY INTO REGISTER M12 CONTENTS OF WORD AT
                         ! LOCATION (PLANE 6, COLUMN 33, WORD 0) IN STORE.
```

RX RXO

Function

RX copies a given store row into an MCU register or the edge register .

RXO copies a given store column into an MCU register or the edge register .

Syntax

RX <MCU-or-edge-register><row><modifier>?<step A>?
RXO <MCU-or-edge-register><column><modifier>?<step A>?

Syntax

where <row>, <modifier> and <step A> together form a store column address, and <column>, <modifier> and <step A> together form a store row address (see section 7.3.3 for details)

Addressing mode

Mode B addressing is used (see section 7.1.2 for details). The effective ADDR field and the effective INT field together define the store row address (for RX) or store column address (for RXO).

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction which specifies the register into which the contents of the addressed row (for RX) or the addressed column (for RXO) are to be loaded

2 *row* specifies the values in the ADDR and INT fields of the instruction, which are used to construct the effective ADDR and INT values

3 *column* specifies the values in the ADDR and INT fields of the instruction, which are used to construct the effective ADDR and INT values

4 *modifier* specifies the value in the MOD field of the instruction which specifies the modifier register to be used, if any

5 *step A* specifies the value in the INCREMENT/DECREMENT field of the instruction, which specifies whether the store address is to be incremented or decremented if it appears in an APAL DO loop. *step A* also specifies the value in the STEP TYPE field of the instruction, which specifies whether the INT or ADDR part of the address is to be stepped

6 If an MCU register is specified then the least significant 32 bits of the row are written into it, and any other bits of the row (for RX) or of the column (for RXO) are discarded

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Examples

```
RX  ME SPLANE  +  10.20  (M3)  (+A)    ! SET THE EDGE REGISTER TO THE
                                       ! CONTENTS OF ROW i OF STORE
                                       ! PLANE j, WHERE:
                                       !
                                       ! i = 20 + (INT FIELD OF M3)
                                       !   + ROW ADDRESS OF 'SPLANE'
                                       !
                                       ! j = THE PLANE ADDRESS OF
                                       ! SPLANE (M3)+10+n, WHERE n
                                       ! IS THE CURRENT DO LOOP ITERATION
                                       ! NUMBER.

RX  M10  GPLANE(M6)                    ! PUT INTO REGISTER M10 THE CONTENTS OF
                                       ! ROW i OF STORE PLANE j; WHERE:
                                       !
                                       ! i = INT FIELD OF M6
                                       !   + ROW ADDRESS OF 'GPLANE'
                                       !
                                       ! j = THE PLANE ADDRESS OF GPLANE
                                       !        + ADDR FIELD OF M6
                                       !
                                       ! THE EFFECT ON DAP 600 WILL BE TO PUT
                                       ! THE 32 LEAST SIGNIFICANT BITS OF THE
                                       ! SELECTED ROW INTO M10.

RXO  ME  26.12  (M2)                   ! SET THE EDGE REGISTER TO THE CONTENTS OF
                                       ! COLUMN i OF STORE PLANE j, WHERE:
                                       !
                                       ! i = 12 + (INT FIELD OF M2)
                                       !
                                       ! j = 26 + (ADDR FIELD OF M2)
```

SAN SAN

Function

SAN copies the inverse of the A plane into a store plane.

Syntax

SAN <plane><modifier>?<step>?

where <plane>, <modifier> and <step> together form a store plane address (see section 7.3.3 for details)

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the SQ instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SAN SPLANE + 9 (M6) (+)     ! SET EACH BIT OF STORE SPLANE (M6) + 9 + i
                            ! (WHERE i IS THE DO LOOP ITERATION NUMBER)
                            ! NUMBER) TO THE INVERSE OF THE CORRESPONDING
                            ! BIT OF THE A PLANE
```

SF

Function

SF sets every bit in a specified store plane to zero.

Syntax

SF <plane><modifier>?<step>?

where <plane>, <modifier> and <step> together form a store plane address (see section 7.3.3 for details)

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the SQ instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SF  27  (M3)                 ! SET EACH BIT OF
                             ! STORE PLANE 27 + (ADDR FIELD OF M3)
                             ! TO ZERO.
```

SHL SHL

Function

SHL copies one MCU register to another, then shifts the latter to the left feeding in zeros on the right. This function may also be used to invoke a single place shift of the edge register.

Syntax

SHL <MCU-register-1><MCU-register-2>?<count>?
SHL ME <count>?

where

> <MCU-register-1> ::= <MCU-register>
> <MCU-register-2> ::= <MCU-register>
> <count> ::= <numval>

Notes

1 *MCU-register-1* specifies the value in the MCUR field of the instruction, which specifies the register to hold the result. If *MCU-register-2* is omitted, this register also holds the operand to be shifted

2 *MCU-register-2* specifies the value in the MOD field of the instruction, and if specified determines the register from which the operand is taken. If *MCU-register-2* is omitted, the MOD field has the same value as the MCUR field

3 *count* specifies the number of places by which *MCU-register-1* is to be shifted. The value of *count* should be in the range 1 to ($ES - 1$); its default value is 1. If the edge register is being shifted, then *count* must either be omitted or specified as 1

Possible run-time program errors

None.

Examples

```
SHL  M1  M2  17        ! SET M1 TO M2 SHIFTED 17 PLACES TO THE LEFT

SHL  M4  12            ! SET M4 TO ITSELF SHIFTED 12 PLACES TO THE LEFT

SHL  ME                ! SHIFT THE EDGE REGISTER 1 PLACE LEFT
```

SHLC SHLC

Function

SHLC copies one MCU register to another, then shifts the latter to the left. Bits shifted off at the left come in at the right. This function may also be used to invoke a single place shift of the edge register.

Syntax

SHLC <MCU-register-1><MCU-register-2>?<count>?
SHLC ME <count>?

where

> <MCU-register-1> ::= <MCU-register>
> <MCU-register-2> ::= <MCU-register>
> <count> ::= <numval>

Notes

See notes for the SHL instruction

Possible run-time program errors

None.

Examples

```
SHLC  M3  14          ! SET M3 TO ITSELF ROTATED 14 PLACES TO THE LEFT

SHLC  M1  M6  7       ! SET M1 TO M6 ROTATED 7 PLACES TO THE LEFT
```

SHR SHR

Function

SHR copies one MCU register to another, then shifts the latter to the right feeding in zeros at the left. This function may also be used to invoke a single place shift of the edge register.

Syntax

```
SHR <MCU-register-1><MCU-register-2>?<count>?
SHR ME <count>?
```

where

```
    <MCU-register-1> ::= <MCU-register>
    <MCU-register-1> ::= <MCU-register>
    <count> ::= <numval>
```

Notes

See notes for the SHL instruction

Possible run-time program errors

None.

Example

```
SHR  M1  M4  12              ! SET M1 TO M4 SHIFTED 12 PLACES TO THE RIGHT
```

SHRA SHRA

Function

SHRA copies one MCU register to another, then shifts the copy to the right. The most significant bit(s) of the copy are filled with repeats of the most significant bit (that is, the sign bit) from the MCU register. This function may also be used to invoke a single place shift of the edge register.

Syntax

SHRA <MCU-register-1><MCU-register-2>?<count>?
SHRA ME <count>?

where

> <MCU-register-1> ::= <MCU-register>
> <MCU-register-1> ::= <MCU-register>
> <count> ::= <numval>

Notes

See notes for the SHL instruction

Possible run-time program errors

None.

Example

```
SHRA  M1  M4  12              ! SET M1 TO M4 SHIFTED 12 PLACES TO THE
                              ! RIGHT, WITH SIGN BIT PROPAGATION
```

SHRC SHRC

Function

SHRC copies one MCU register to another, then shifts the latter to the right. Bits shifted off at the right come in at the left. This function may also be used to invoke a single place shift of the edge register.

Syntax

SHRC <MCU-register-1><MCU-register-2>?<count>?
SHRC ME <count>?

where

    <MCU-register-1> ::= <MCU-register>
    <MCU-register-2> ::= <MCU-register>

<count> ::= <numval>

Notes

See notes for the SHL instruction

Possible run-time program errors

None.

Example

```
SHRC  M1  10                ! SET M1 TO ITSELF ROTATED 10 PLACES TO THE
                            ! RIGHT
```

SIC SIC

Function

SIC copies the C plane into a given store plane, under activity control (see section 1.6 for details).

Syntax

SIC <plane><modifier>?<step>?

where <plane>, <modifier> and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

1 *plane* specifies the value in the ADDR field of the instruction, which is used to construct the effective ADDR value

2 *modifier* specifies the value in the MOD field of the instruction which specifies the modifier register to be used, if any

3 *step* specifies the value in the INCREMENT/DECREMENT field of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop

4 The only bits of the store plane which are updated are those corresponding to those bits in the A plane that are true (that is, are set to one)

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SIC  SPLANE  (M4)  (+)          ! SET EACH BIT OF STORE PLANE SPLANE (M4) + i
                                ! (WHERE i IS THE DO LOOP ITERATION NUMBER)
                                ! CORRESPONDING TO A 'ONE' BIT IN THE A PLANE
                                ! TO THE CORRESPONDING BIT OF PLANE C. LEAVE
                                ! OTHER BITS OF STORE SPLANE (M4) + i
                                ! UNCHANGED
```

SICPCQS SICPCQS

Function

SICPCQS adds corresponding bits of the C plane, Q plane and a store plane putting the sum in the store plane, under activity control (see section 1.6)

Carry bits go into the C plane (in every PE).

Syntax

SICPCQS <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the SIC instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SICPCQS  28  (M4)          ! SET EACH BIT OF STORE PLANE
                           ! 28 + (ADDR FIELD OF M4)
                           ! THAT CORRESPONDS TO A 'ONE' BIT IN THE
                           ! A PLANE TO THE SUM OF ITSELF AND THE
                           ! CORRESPONDING BITS OF THE C AND Q PLANES.
                           ! LEAVE OTHER BITS OF THE STORE PLANE
                           ! UNCHANGED.
                           !
                           ! IRRESPECTIVE OF THE VALUE OF THE BITS
                           ! IN THE A PLANE SET EACH BIT OF THE
                           ! C PLANE TO THE CARRY BIT GENERATED BY
                           ! THE CORRESPONDING ADDITION.
```

SICPCS SICPCS

Function

SICPCS adds corresponding bits of the C plane and a store plane, putting the sum in the store plane, under activity control (see section 1.6 for details)

Carry bits go into the C plane (in every PE).

Syntax

SICPCS <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the SIC instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SICPCS  SPLANE + 9 (M4) (+)  ! SET EACH BIT OF STORE PLANE
                             ! SPLANE (M4) + 9 + i
                             ! (WHERE i IS THE DO LOOP ITERATION NUMBER)
                             ! THAT CORRESPONDS TO A 'ONE' IN
                             ! THE A PLANE TO THE SUM OF ITSELF AND
                             ! THE CORRESPONDING BIT OF THE C PLANE.
                             ! LEAVE OTHER BITS OF THE STORE PLANE
                             ! UNCHANGED.
                             !
                             ! IRRESPECTIVE OF THE VALUE OF THE BITS
                             ! IN THE A PLANE SET EACH BIT OF THE
                             ! C PLANE TO THE CARRY BIT GENERATED BY
                             ! THE CORRESPONDING ADDITION.
```

SICPQS SICPQS

Function

SICPQS adds corresponding bits of the Q plane and a store plane, putting the sum in the store plane, under activity control (see section 1.6 for details)

Carry bits go into the C plane (in every PE).

Syntax

SICPQS <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the SIC instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SICPQS  28  (-)          ! SET EACH BIT OF STORE PLANE 28 - i
                         ! (WHERE i IS THE DO LOOP ITERATION NUMBER)
                         ! THAT CORRESPONDS TO A 'ONE BIT' IN THE
                         ! A PLANE TO THE SUM OF ITSELF AND THE
                         ! CORRESPONDING BIT OF THE Q PLANE. LEAVE
                         ! OTHER BITS OF THE STORE PLANE UNCHANGED.
                         !
                         ! IRRESPECTIVE OF VALUE OF CORRESPONDING
                         ! BITS IN PLANE A, SET EACH BIT OF PLANE C
                         ! TO CARRY BIT GENERATED BY CORRESPONDING
                         ! ADDITIONS.
```

SICQPCQS

Function

SICQPCQS adds corresponding bits of the C plane, Q plane and a store plane putting the sum in the Q plane. It also puts the sum in the store plane but under activity control (see section 1.6 for details).

Carry bits go into the C plane (in every PE).

Syntax

SICQPCQS <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the SIC instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SICQPCQS  39  (M4)          ! SET EACH BIT OF STORE PLANE
                            ! 39 + (ADDR FIELD OF M4)
                            ! THAT CORRESPONDS TO A 'ONE' BIT IN THE
                            ! A PLANE TO THE SUM OF ITSELF AND THE
                            ! CORRESPONDING BITS OF THE C AND Q PLANES.
                            ! LEAVE OTHER BITS OF THE STORE PLANE
                            ! UNCHANGED.
                            !
                            ! IRRESPECTIVE OF THE VALUE OF THE BITS IN
                            ! THE A PLANE, SET THE SUMS OF ALL THE
                            ! CORRESPONDING BITS ORIGINALLY IN
                            ! PLANES C, Q AND 39+(ADDR FIELD OF M4)
                            ! INTO THE CORRESPONDING BITS IN THE
                            ! Q PLANE AND SET THE CARRY BITS INTO THE
                            ! CORRESPONDING BITS OF THE C PLANE.
```

SICQPCS SICQPCS

Function

SICQPCS adds corresponding bits of the C plane and a store plane, putting the sum in the Q plane. It also puts the sum in the store plane, but under activity control (see section 1.6 for details).

Carry bits go into the C plane (in every PE).

Syntax

SICQPCS <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the SIC instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SICQPCS 53 (+)          ! SET EACH BIT OF STORE PLANE 53 + i
                        ! (WHEREi IS THE DO LOOP ITERATION NUMBER)
                        ! THAT CORRESPONDS TO A 'ONE' BIT IN THE
                        ! A PLANE TO THE SUM OF ITSELF AND THE
                        ! CORRESPONDING BIT OF THE C PLANE. LEAVE
                        ! OTHER BITS OF THE STORE PLANE UNCHANGED.
                        !
                        ! IRRESPECTIVE OF THE VALUE OF THE BITS IN
                        ! THE A PLANE, SET THE SUMS OF THE
                        ! CORRESPONDING BITS ORIGINALLY IN
                        ! PLANES C AND 53 + i INTO THE BITS IN
                        ! THE Q PLANE, AND THE CARRY BITS INTO
                        ! THE CORRERSPONDING BITS OF THE C PLANE.
```

SICQPQS SICQPQS

Function

SICQPQS adds corresponding bits of the Q plane and a store plane, putting the sum in the Q plane. It also puts the sum in the store plane, but under activity control (see section 1.6 for details).

Carry bits go into the C plane (in every PE).

Syntax

SICQPQS <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the SIC instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SICQPQS  SPLANE  (M6)        ! SET EACH BIT OF SPLANE (M6)
                             ! CORRESPONDING TO A 'ONE' BIT IN THE A PLANE
                             ! TO THE SUM OF ITSELF AND THE
                             ! CORRESPONDING BIT OF THE Q PLANE. LEAVE
                             ! OTHER BITS OF THE STORE PLANE UNCHANGED.
                             !
                             ! IRRESPECTIVE OF THE VALUE OF THE BITS IN
                             ! THE A PLANE, SET THE SUMS OF THE
                             ! CORRESPONDING BITS ORIGINALLY IN
                             ! INTO THE BITS IN THE Q PLANE, AND SET THE
                             ! PLANES Q AND SPLANE (M6) CARRY BITS INTO
                             ! THE CORRESPONDING BITS OF THE C PLANE.
```

SIF

Function

SIF sets every bit of a given store plane to zero, under activity control (see section 1.6 for details).

Syntax

SIF <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the SIC instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SIF  SPLANE + 3 (M2)       ! SET EACH BIT OF PLANE SPLANE (M2) + 3
                           ! THAT CORRESPONDS TO A 'ONE' BIT IN
                           ! THE A PLANE TO ZERO. LEAVE OTHER BITS OF
                           ! THE STORE PLANE UNCHANGED.
```

SIPCQ SIPCQ

Function

SIPCQ adds corresponding bits of the C and Q planes, putting the sum in a store plane under activity control (see section 1.6 for details)

Carry bits are discarded.

Syntax

SIPCQ <plane><modifier>?<step>?

where <plane>, <modifier> and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the SIC instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SIPCQ  10                ! SET EACH BIT OF PLANE 10 THAT
                         ! CORRESPONDS TO A 'ONE' BIT IN THE A PLANE
                         ! TO THE SUM OF THE CORRESPONDING BITS
                         ! IN THE C AND Q PLANES. LEAVE OTHER BITS OF
                         ! PLANE 10 UNCHANGED.
```

SIPCQS SIPCQS

Function

SIPCQS adds corresponding bits of the C plane, Q plane and a store plane putting the sum in the store plane under activity control (see section 1.6)

Carry bits are discarded.

Syntax

SIPCQS <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the SIC instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SIPCQS  SPLANE  (M3)          ! SET EACH BIT OF SPLANE (M3) THAT
                              ! CORRESPONDS TO A 'ONE' BIT IN THE A PLANE
                              ! TO THE SUM OF ITSELF AND THE
                              ! CORRESPONDING BITS OF THE C AND Q PLANES.
                              ! LEAVE OTHER BITS OF SPLANE (M3) UNCHANGED.
```

SIPCS SIPCS

Function

SIPCS adds corresponding bits of a store plane and the C plane, putting the sum in the store plane under activity control (see section 1.6 for details)

Carry bits are discarded.

Syntax

SIPCS <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the SIC instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SIPCS  SPLANE + 1 (M4)     ! SET EACH BIT OF SPLANE (M4) + 1
                           ! THAT CORRESPONDS TO A 'ONE' BIT IN
                           ! THE A PLANE TO THE SUM OF ITSELF AND
                           ! THE CORRESPONDING BIT OF THE C PLANE.
                           ! LEAVE OTHER BITS IN SPLANE (M4) + 1
                           ! UNCHANGED.
```

SIPQS SIPQS

Function

SIPQS adds corresponding bits of a store plane and the Q plane, putting the sum in the store plane, under activity control (see section 1.6 for details)

Carry bits are discarded.

Syntax

SIPQS <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the SIC instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SIPQS SPLANE (M4) (+)       ! SET EACH BIT OF SPLANE (M4) + i
                            ! (WHERE i IS THE CURRENT DO LOOP STEP
                            ! VALUE) THAT CORRESPONDS TO A 'ONE' BIT
                            ! IN THE A PLAN TO THE SUM OF ITSELF AND
                            ! THE CORRESPONDING BIT OF PLANE Q.
                            ! LEAVE OTHER BITS OF SPLANE (M4) + i
                            ! UNCHANGED.
```

SIQ SIQ

Function

SIQ copies the Q plane to a store plane, under activity control (section 1.6).

Syntax

SIQ <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the SIC instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SIQ  SPLANE  (M4)          ! SET EACH BIT OF SPLANE (M4) THAT
                           ! CORRESPONDS TO A 'ONE' BIT IN THE A PLANE
                           ! TO THE CORRESPONDING BIT OF THE Q PLANE.
                           ! LEAVE OTHER BITS OF SPLANE (M4)
                           ! UNCHANGED.
```

SIQPCQS SIQPCQS

Function

SIQPCQS adds corresponding bits of the C plane, Q plane and a store plane putting the sum in the Q plane. It also puts the sum in the store plane but under activity control (see section 1.6 for details).

Carry bits are discarded.

Syntax

SIQPCQS <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the SIC instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SIQPCQS SPLANE + 5 (M7)  ! SET EACH BIT OF SPLANE (M7) + 5
                         ! THAT CORRESPONDS TO A 'ONE' BIT
                         ! IN THE A PLANE TO THE SUM OF ITSELF AND
                         ! THE CORRESPONDING BITS IN THE C AND Q
                         ! PLANES. LEAVE OTHER BITS IN THE STORE
                         ! PLANE UNCHANGED.
                         !
                         ! IRRESPECTIVE OF THE VALUE OF BITS IN
                         ! THE PLANE A, SET ALL BITS OF THE Q PLANE
                         ! TO THE SUMS OF THE CORRESPONDING BITS IN
                         ! THE STORE PLANE AND THE C AND Q PLANES.
```

SIQPCS SIQPCS

Function

SIQPCS adds corresponding bits of a store plane and the C plane, putting the result in the Q plane. It also puts the result in the store plane, but under activity control (see section 1.6 for details).

Carry bits are discarded.

Syntax

SIQPCS <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the SIC instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SIQPCS  SPLANE  (M4)        ! SET EACH BIT OF SPLANE (M4)
                            ! THAT CORRESPONDS TO A 'ONE' BIT IN
                            ! THE A PLANE TO THE SUM OF THE
                            ! CORRESPONDING BITS OF ITSELF AND
                            ! THE C PLANE. LEAVE OTHER BITS OF
                            ! SPLANE (M4) UNCHANGED.
                            !
                            ! IRRESPECTIVE OF THE VALUE OF BITS
                            ! IN THE A PLANE, SET ALL BITS IN THE
                            ! Q PLANE TO THE SUMS OF CORRESPONDING
                            ! BITS ORIGINALLY IN PLANES C AND
                            ! SPLANE (M4).
```

SIQPQS SIQPQS

Function

SIQPQS adds corresponding bits of a store plane and the Q plane, putting the sum in the Q plane. It also puts the sum in the store plane, under activity control (see section 1.6 for details).

Carry bits are discarded.

Syntax

SIQPQS <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the SIC instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SIQPQS  12                ! SET EACH BIT OF PLANE 12 THAT
                          ! CORRESPONDS TO A 'ONE' BIT IN
                          ! THE A PLANE TO THE SUM OF THE
                          ! CORRESPONDING BITS IN PLANE 12
                          ! AND THE Q PLANE. LEAVE OTHER BITS
                          ! IN PLANE 12 UNCHANGED.
                          !
                          ! IRRESPECTIVE OF THE VALUE OF THE
                          ! BITS IN THE A PLANE, SET ALL BITS
                          ! OF THE Q PLANE WITH THE SUMS OF
                          ! THE CORRESPONDING BITS ORIGINALLY
                          ! IN ITSELF AND PLANE 12.
```

SKIP SKIP

Function

SKIP skips (that is, ignores) the next instruction if one or all bits of a specified MCU register or the edge register have a certain value, or if the CARRY or V flag has a certain value.

Syntax

SKIP <MCU-or-edge-register>.<bit number><modifier>?<step>?<truth value>
SKIP <MCU-or-edge-register> ALL <truth value>
SKIP <MCU-or-edge-register> ANY <truth value>
SKIP C <truth value>
SKIP V <truth value>

where<MCU-or-edge-register>, <bit number>, <modifier>, and <step> together form a register bit address (see section 7.3.4 for details).

<truth value> ::= 0 | 1 | T | F

The truth values 0 and F are equivalent, as are the values 1 and T.

Addressing mode

Mode A addressing is used for the form of SKIP first in the list above (see section 7.1.1 for details). The effective INT field defines the bit number of the MCU register or the edge register, and the effective ADDR field is discarded. The other forms of SKIP do not use addressing.

Notes

1 *MCU-or-edge-register* specifies the value in the MCUR field of the instruction which specifies the register whose bit(s) are to be tested by this instruction

2 *bit-number* specifies the value in the INT field of the instruction which is used to construct the effective INT value

3 *modifier* specifies the value in the MOD field of the instruction which specifies the modifier register to be used, if any. If modifier is omitted, the MOD field is set to zero

4 *step* specifies the value in the INCREMENT/DECREMENT field of the instruction, which specifies how the bit number is to be stepped if the instruction appears inside an APAL DO loop

5 The first form of SKIP tests an individual bit of an MCU register or of the edge register. If the bit specified by the effective INT field has the same value as *truth value*, then the next APAL instruction is skipped

6 If an individual MCU register bit is being tested, then the effective INT field modulo 32 is used; if an individual bit in the edge register is being tested, then the full effective INT field is used

SKIP SKIP

Notes – continued

7 If SKIP ALL is specified, then the next instruction is skipped only if all bits in the specified register have the same value as *truth value*

8 If SKIP ANY is specified, then the next instruction is skipped if any bits in the specified register have the same value as *truth value*

9 If SKIP C is specified, then the next instruction is skipped if *truth value* and the CARRY flag have the same value; the CARRY flag is not changed by SKIP C

10 If SKIP V is specified, then the next instruction is skipped if *truth value* and the V flag have the same value; the V flag is not changed by SKIP V

11 If SKIP is the last instruction in a DO loop, the effect is undefined

Possible run-time program errors

None

Examples

```
SKIP  M4.12  F               ! SKIP THE NEXT INSTRUCTION IF BIT 12
                             ! OF M4 IS ZERO.

SKIP  ME.3(M5)  1            ! SKIP THE NEXT INSTRUCTION IF BIT
                             ! (3 + INT FIELD OF M5) OF ME IS .TRUE.

SKIP  M1  ALL  0             ! SKIP THE NEXT INSTRUCTION IF EVERY BIT
                             ! OF M1 IS 0.

SKIP  M4  ANY  T             ! SKIP THE NEXT INSTRUCTION IF ANY BIT
                             ! OF M4 IS 1.

ADD  M1  M2                  ! SKIP THE NEXT INSTRUCTION IF THERE WAS
SKIP  C  F                   ! NO CARRY-OUT OF THE MOST SIGNIFICANT BIT
                             ! POSITION IN THE ADD INSTRUCTION

ADD  M1  M2                  ! SKIP THE NEXT INSTRUCTION IF THERE WAS
SKIP  V  F                   ! NO OVERFLOW IN THE ADD INSTRUCTION
```

SQ SQ

Function

The SQ instruction copies the Q plane into a given store plane.

Syntax

SQ <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

1 *plane* specifies the value in the ADDR field of the instruction, which is used to construct the effective ADDR value

2 *modifier* specifies the value in the MOD field of the instruction which specifies the modifier register to be used, if any

3 *step* specifies the value in the INCREMENT/DECREMENT field of the instruction, which specifies how the store plane address is to be stepped if the instruction appears inside an APAL DO loop

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SQ  22  (M6)  (+)          ! SET EACH BIT OF STORE PLANE
                           ! 22 + (ADDR FIELD OF M6) + i
                           ! (WHERE i IS THE DO LOOP ITERATION NUMBER)
                           ! TO THE CORRESPONDING BIT OF THE Q PLANE.
```

SQ_AQ SQ_AQ

Function

SQ_AQ is a compound instruction. It copies the Q plane into a given store plane and into the A plane.

Syntax

SQ_AQ <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the SQ instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SQ_AQ  0  (M6)                ! SET EACH BIT OF STORE PLANE (ADDR FIELD
                              ! OF M6) AND THE A PLANE TO THE CORRESPONDING
                              ! BIT OF THE Q PLANE.
```

SQ_CQ SQ_CQ

Function

SQ_CQ is a compound instruction. It copies the Q plane into a given store plane and into the C plane.

Syntax

SQ_CQ <plane><modifier>?<step>?

where <plane>, <modifier> and <step> form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the SQ instruction

Possible run-time program errors

Example

```
SQ_CQ  10                    ! SET EACH BIT OF PLANE 10 AND
                             ! THE C PLANE TO THE CORRESPONDING BIT OF
                             ! THE Q PLANE
```

SQ_QC SQ_QCN

Function

SQ_QC is a compound instruction. It copies the Q plane into a given store plane and then copies the C plane into the Q plane.

SQ_QCN copies the Q plane into a given store plane, then copies the inverse of the C plane into the Q plane.

Syntax

SQ_QC <plane><modifier>?<step>?
SQ_QCN <plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the SQ instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SQ_QC SPLANE  (M2)  (-)      ! SET EACH BIT OF STORE PLANE
                             ! SPLANE (M2) - i (WHERE i IS THE DO LOOP
                             ! ITERATION NUMBER) TO THE CORRESPONDING
                             ! BIT OF THE Q PLANE.
                             !
                             ! THEN SET EACH BIT OF THE Q PLANE TO
                             ! THE CORRESPONDING BIT OF THE C PLANE.
```

SQ_QF SQ_QT

Function

SQ_QF is a compound instruction. It copies the Q plane to a given store plane then sets every bit of the Q plane to zero.

SQ_QT copies the Q plane to a given store plane, then sets every bit of the Q plane to one.

Syntax

```
SQ_QF <plane><modifier>?<step>?
SQ_QT <plane><modifier>?<step>?
```

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details)

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

See notes for the SQ instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SQ_QT  SPLANE  (M5-)        ! SET EACH BIT OF STORE PLANE
                            ! SPLANE (M5) – i (WHERE i IS THE DO LOOP
                            ! ITERATION NUMBER) TO THE CORRESPONDING
                            ! BIT OF THE Q PLANE.
                            !
                            ! THEN SET EACH BIT OF THE Q PLANE TO ONE.
```

SR SRN

Function

SR copies the R plane into a given store plane.

SRN copies the inverse of the R plane into a given store plane.

Syntax

SR <MCU-or-edge-register><plane><modifier>?<step>?
SRN <MCU-or-edge-register><plane><modifier>?<step>?

where <plane>, <modifier>, and <step> together form a store plane address (see section 7.3.3 for details).

Addressing mode

Mode A addressing is used (see section 7.1.1 for details). The effective ADDR field defines the store plane address, and the effective INT field is discarded.

Notes

1 to 4	See notes for the SQ instruction
5	*MCU-or-edge-register* specifies the value in the MCUR field of the instruction which specifies the register that is used to form the R plane (see section 6.2 for details)

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
SR  ME  SPLANE  (M2)         ! SET EACH ROW OF STORE PLANE SPLANE (M2)
                             ! TO THE CONTENTS OF THE EDGE REGISTER.
```

STOP STOP

Function

STOP is a pseudo instruction that causes the DAP program and its associated host program to be abandoned, and a message to be output via the run-time support system; it is one of several instructions (PAUSE, STOP and TRACE) that the assembler encodes as particular cases of the CALL instruction. STOP is normally used to indicate error or exception conditions. For further details see *DAP Series: Program Development*.

Syntax

STOP <error number>?

where <error number> ::= <numval>

Notes

1. *error-number* must be in the range 1 to 262,143
2. *error-number* is printed as part of the message output by the run-time support system when STOP is obeyed
3. STOP causes a user defined APAL error. To generate a non-error exit, the EXIT instruction should be used. An EXIT in the top level DAP subroutine will cause return to the calling host program. See chapter 9 for further details

Possible run-time program errors

None.

Example

```
STOP  2000                    ! ABANDON THE PROGRAM, REPORTING THE USER
                              ! DEFINED APAL ERROR NUMBER OF 2000.
```

SUB SUB

Function

SUB subtracts one MCU register from another. It also assigns the CARRY and OFLO flags.

Syntax

SUB <MCU-register-1><MCU-register-2>

where

```
<MCU-register-1> ::= <MCU-register>
<MCU-register-2> ::= <MCU-register>
```

Notes

1 *MCU-register-1* specifies the value in the MCUR field of the instruction, which specifies both the register containing the first operand and the register to contain the result

2 *MCU-register-2* specifies the value in the MOD field of the instruction, which specifies the register containing the second operand

3 The CARRY and OFLO flags are assigned as described in section 1.10

Possible run-time program errors

None.

Example

```
SUB  M3  M4                  ! LOAD M3 WITH M3 – M4. SET THE CARRY FLAG
                             ! IF M3 >= M4
```

SUBC

Function

SUBC subtracts one MCU register from another, using the CARRY flag as the inverse of borrow-in at the least significant bit. It also assigns the CARRY and OFLO flags.

Syntax

SUBC <MCU-register-1><MCU-register-2>

where

```
<MCU-register-1> ::= <MCU-register>
<MCU-register-2> ::= <MCU-register>
```

Notes

1 *MCU-register-1* specifies the value in the MCUR field of the instruction, which specifies both the register containing the first operand and the register to contain the result

2 *MCU-register-2* specifies the value in the MOD field of the instruction, which specifies the register containing the second operand

3 The CARRY and OFLO flags are assigned as described in section 1.10

Possible run-time program errors

None.

Example

```
SUB  M2  M10            ! SUBTRACT THE 64-BIT NUMBER IN REGISTER
SUBC  M3  M12           ! PAIR M10 AND M12 FROM THE 64-BIT NUMBER
                        ! IN REGISTER PAIR M2 AND M3. PUT THE RESULT
                        ! IN M2, M3
```

SUBH SUBH

Function

SUBH subtracts a literal from an MCU register. It also assigns the CARRY and V flags.

Syntax

SUBH <MCU-register><literal_16>

where <literal_16> is defined in section 6.1.7.

Notes

1 *MCU-register* specifies the value in the MCUR field of the instruction which specifies both the register containing the first operand and the register to contain the result

2 *literal_16* specifies the value in the LITERAL field. To form the second operand, the bit pattern representing the value is expanded to 32 bits with leading zeros (see section 6.1.7 for details for more details)

3 The CARRY and OFLO flags are assigned as described in section 1.10

Possible run-time program errors

None

Example

```
SUBH  M1  #FF                   ! LOAD M1 WITH M1-#000000FF. SET THE CARRY FLAG
                                ! IF M1 >= #FF
                                !
                                ! SET V IF M1 CHANGES FROM NEGATIVE TO POSITIVE
```

SUBHC SUBHC

Function

SUBHC subtracts a literal from an MCU register, using the CARRY flag as the inverse of borrow-in at the least significant bit. It also assigns the CARRY and OFLO flags.

Syntax

SUBHC <MCU-register><literal_16>

where <literal_16> is defined in section 6.1.7.

Notes

1 *MCU-register* specifies the value in the MCUR field of the instruction which specifies both the register containing the first operand and the register to contain the result

2 *literal_16* specifies the value in the LITERAL field. To form the second operand, the bit pattern representing the value is expanded to 32 bits with leading zeros (see section 6.1.7 for details for more details)

3 The CARRY and OFLO flags are assigned as described in section 1.10

Possible run-time program errors

None

Example

```
SUBH  M5  27               ! SUBTRACT THE LITERAL VALUE 27 FROM THE
SUBHC  M4  0               ! 64-BIT NUMBER HELD IN THE REGISTER PAIR
                           ! M4, M5. PUT THE RESULT IN M4, M5
```

TRACE TRACE

Function

TRACE suspends the execution of an APAL program temporarily, while it outputs the contents of the requested registers and array store locations.

Syntax

```
TRACE<trace_number>?<registers_trace_item><trace_level><newline> |
TRACE<trace_number>?<registers_trace_item>?<trace_level><array_store_trace_item><newline>
```

where:

```
<array_store_trace_item> ::= <word><modifier>?<trace_count>? WORDPACK?<type/size>?|
                             <word><modifier>?<trace_count>? ROWPACK<type/size>?
                                     <start_bit>?|
                             <word><modifier>?<trace_count>? VERTICAL<type/size>?
                                     <row_range>?<col_range>?
                             <word><modifier>?<trace_count>? VERTICAL<type/size>?
                                     <col_range>?<row_range>?
```

Notes

1 <trace_number> is a number that will be printed at the head the TRACE output; it defaults to the line number of the TRACE statement, as given in the listing of the APAL source code containing the TRACE statement

2 <registers_trace_item> specifies which of the registers are to be TRACEd. Options are MER (the contents of the MCU and edge registers); PER (the A, Q and C planes of the PEs); or MER PER or PER MER (both have the same effect: PER then MER)

3 <trace_level> has the form 'LEVEL <numval>', where <numval> is in the range 1 to 15. The assembler will not generate code for a TRACE statement if <trace_level> exceeds the 'assembly-trace-level' parameter; an assembled TRACE statement will not generate output if <trace_level> exceeds the 'run-time-trace-level' parameter

4 <trace_count> is a number specifying the number of store items to be TRACEd; the default is 1. Where WORDPACK or ROWPACK is specified, each item TRACEd is one scalar value; where VERTICAL, each item is a matrix of values

5 <type/size> specifies the way in which the items to be TRACEd are to be assumed to be mapped in memory (options are HEX, INT, REAL, CHAR and BIT, with a default of HEX), and the assumed size in bits of the item, with a default of 32; <size> takes the form (*numval*). Neither <type> nor <size> is required; <type> can be specified without <size>, but not vice versa

6 <start_bit> specifies an offset in bits from the start (MSB) of the row, of the data item(s) to be TRACEd, and takes the form 'FROM_BIT <numval>'. The default is such that each item is assumed to be 'right aligned' in its row; for example, in code to run on a DAP 600, if <type/size> is CHAR (24), then the default for <start_bit> is 40

Syntax – continued

7 <row_range> specifies the range of rows (or part rows) to be TRACEd, and takes the form 'ROWS (<numval>,<numval>)', where <numval> is in the range 0 to $ES - 1$; it defaults to all rows

8 <col_range> specifies the range of columns (or part columns) to be TRACEd, and takes the form 'COLS (<numval>,<numval>)', where <numval> is in the range 0 to $ES - 1$; it defaults to all columns

See chapter 8 for more details.

Addressing mode

A reduced form of Mode C addressing is used to specify the array store items to be TRACEd:

<word><modifier>? specifies the starting address in store of the item(s). If <modifier> is present, then the contents of the modifier register (one of M1 to M7) are addded to the word address, and the resultant sum used to define the starting word address.

If ROWPACK or VERTICAL format is specified, then TRACEing starts respectively at the row or plane containing that starting word address (modified by any FROM_BIT or COLS/ROWS values, if specified)

See chapter 8 for more details.

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

See the next page for examples of TRACE.

TRACE TRACE

Examples

```
MYDATA = 25.34.1

.
.
.

TRACE 2 LEVEL 10 MYDATA #100 WORDPACK HEX (8)

                                ! OUTPUT TRACE REPORT NUMBER 2, IF TRACES OF
                                ! LEVEL 10 OR LOWER ARE REQUIRED.

                                ! THE DATA ITEMS TO BE TRACED START AT
                                ! THE LOCATION LABELLED 'MYDATA'
                                ! (PLANE 25 ROW 34 WORD 1);
                                ! 100 (IN HEX) DATA ITEMS WILL BE TRACED.

                                ! EACH DATA ITEM IS PACKED (RIGHT JUSTIFIED)
                                ! INTO A WORD; IS PRINTED IN HEX FORMAT; IS TAKEN
                                ! FROM THE LEAST SIGNIFICANT 8 BITS OF THE WORD.
.
.
.

TRACE 6 MER PER LEVEL 3 MYDATA(M5) VERTICAL CHAR (8) ROWS(0,3) COLS(0,5)

                                ! OUTPUT TRACE REPORT NUMBER 6, IF TRACES
                                ! OF LEVEL 3 OR LOWER ARE REQUIRED. TRACE
                                ! THE MCU, EDGE AND PE REGISTERS, AND STORE ITEMS.

                                ! THE STORE ITEMS TO BE TRACED START AT
                                ! LOCATION LABELLED 'MYDATA', MODIFIED BY THE
                                ! CONTENTS OF M5; THAT IS AT THE PLANE GIVEN BY THE
                                ! ADDR COMPONENT OF (25.34.1 + CONTENTS OF M5).

                                ! 1 ITEM ONLY WILL BE OUTPUT; IT IS ASSUMED TO BE
                                ! HELD IN VERTICAL FORMAT, TO BE OF TYPE CHARACTER,
                                ! AND TO BE A SERIES OF SINGLE CHARACTERS:
                                ! THE 24 CHARACTERS STORED IN THE AREA BOUNDED
                                ! BY ROWS 0 TO 3 AND COLUMNS 0 TO 5 OF THE
                                ! SELECTED 8 PLANES ARE TRACED.
```

WF WF

Function

WF sets every bit in a specified word of store to zero.

Syntax

WF <word><modifier>?<step>?

where <word>, <modifier>, and <step> together form a store word address (see section 7.3.3 for details).

Addressing mode

Mode C addressing is used (see section 7.1.3 for details for details). The effective ADDR, INT and (where applicable) WORD fields together define the store word address.

Notes

1 *word* specifies the values in the ADDR, INT and WORD fields of the instruction, which are used to construct the effective ADDR, INT and WORD values

2 *modifier* specifies the value in the MOD field of the instruction which specifies the modifier register to be used, if any

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
WF  18..4                        ! SET WORD 4 OF STORE PLANE 18 TO ALL ZEROS
```

WR WRN

Function

WR copies a given MCU register into a given store word.

WRN copies the inverse of a given MCU register into a given store word.

Syntax

WR <MCU-register><word><modifier>?<step>?
WRN <MCU-register><word><modifier>?<step>?

where <word>, <modifier>, and <step> together form a store word address (see section 7.3.3 for details).

Addressing mode

Mode C addressing is used (see section 7.1.3 for details for details). The effective ADDR, INT and (where applicable) WORD fields together define the store word address.

Notes

1 *word* specifies the values in the ADDR, INT and WORD fields of the instruction, which are used to construct the effective ADDR, INT and WORD values

2 *modifier* specifies the value in the MOD field of the instruction which specifies the modifier register to be used, if any

3 *MCU-register* specifies the value in the MCUR field of the instruction which specifies the register to be copied

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
WR  M4  28..10  (-)          ! COPY THE CONTENTS OF M4 INTO THE WORD
                             ! WHOSE ADDRESS IS 28*ES²/32 + 10 - i, WHERE
                             ! i IS THE DO LOOP ITERATION NUMBER
```

XAN XAN

Function

XAN sets a specified row of store to the inverse of the corresponding row of the A plane.

Syntax

XAN <row><modifier>?<step A>?

where <row>, <modifier>, and <step A> together form a store row address (see section 7.3.3 for details).

Addressing mode

Mode B addressing is used (see section 7.1.2 for details). The effective ADDR field and the effective INT field together define the store row address.

Notes

1 *row* specifies the values in the ADDR and INT fields of the instruction, which are used to construct the effective ADDR and INT values

2 *modifier* specifies the value in the MOD field of the instruction which specifies the modifier register to be used, if any

3 *step A* specifies the value in the INCREMENT/DECREMENT field of the instruction, which determines whether the data address is to be incremented or decremented if the instruction appears in an APAL DO loop. *step A* also specifies the value in the STEP TYPE field of the instruction, which determines whether the INT or ADDR part of the address is to be stepped

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
XAN  12.9  (M4)              ! SET ROW 9 + (INT FIELD OF M4) OF STORE PLANE
                             ! 12 + (ADDR FIELD OF M4) + CARRY-OUT OF INT FIELD
                             ! TO THE CORRESPONDING ROW OF THE INVERSE OF
                             ! THE A PLANE
```

XF XF

Function

XF sets every bit in a specified row of store to zero.

Syntax

XF <row><modifier>?<step A>?

where <row>, <modifier>, and <step A> together form a store row address (see section 7.3.3 for details).

Addressing mode

Mode B addressing is used (see section 7.1.2 for details). The effective ADDR field and the effective INT field together define the store row address.

Notes

See notes for the XAN instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
XF  18.4                              ! SET ROW 4 OF STORE PLANE 18 TO ALL ZEROS
```

XIC XIC

Function

XIC sets a given store row to the corresponding C plane row, under activity control (see section 1.6 for details).

Syntax

XIC <row><modifier>?<step A>?

where <row>, <modifier>, and <step A> together form a store row address (see section 7.3.3 for details).

Addressing mode

Mode B addressing is used (see section 7.1.2 for details). The effective ADDR field and the effective INT field together define the store row address.

Notes

1 to 3	See notes for the XAN instruction
4	The only bits of the store row that are updated are the ones corresponding to those bits in the A plane that are true

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
XIC  SPLANE  +  2.12  (M4)     ! SET EACH BIT OF THE SELECTED
                               ! STORE ROW TO THE CORRESPONDING BIT OF
                               ! THE C PLANE, BUT ONLY WHEN THE
                               ! CORRESPONDING BIT OF THE A PLANE IS TRUE
                               !
                               ! THE STORE ROW ADDRESS IS CONSTRUCTED
                               ! USING MODE B ADDRESSING
```

XIF XIF

Function

XIF sets a given store row to zero, under activity control (see section 1.6).

Syntax

XIF <row><modifier>?<step A>?

where <row>, <modifier>, and <step A> together form a store row address (see section 7.3.3 for details).

Addressing mode

Mode B addressing is used (see section 7.1.2 for details). The effective ADDR field and the effective INT field together define the store row address.

Notes

See notes for the XIC instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
XIF  21.10                ! SET EACH BIT OF ROW 10 OF STORE PLANE 21
                          ! TO ZERO WHEREVER THE CORRESPONDING BIT IN
                          ! ROW 10 OF THE A PLANE IS ONE.
```

XIPCQ XIPCQ

Function

XIPCQ sets a given store row to the corresponding row extracted from the sum of the C plane and the Q plane, under activity control (see section 1.6 for details).

Syntax

XIPCQ <row><modifier>?<step A>?

where <row>, <modifier>, and <step A> together form a store row address (see section 7.3.3 for details).

Addressing mode

Mode B addressing is used (see section 7.1.2 for details). The effective ADDR field and the effective INT field together define the store row address.

Notes

1 to 3	See notes for the XAN instuction
4	The only bits of the store row that are updated are the ones corresponding to those bits in the A plane that are true

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
XIPCQ  SPLANE + 2.12 (M4)      ! SET EACH BIT OF A STORE ROW TO THE
                               ! CORRESPONDING BIT OF THE EXCLUSIVE-OR OF
                               ! THE C PLANE WITH THE Q PLANE, BUT ONLY
                               ! WHERE THE CORRESPONDING BIT OF THE A PLANE
                               ! IS TRUE.
                               !
                               ! THE STORE ROW ADRESS IS CONSTRUCTED USING
                               ! MODE B ADDRESSING.
```

XIQ XIQ

Function

XIQ sets a given store row to the corresponding Q plane row, under activity control (see section 1.6 for details).

Syntax

XIQ <row><modifier>?<step A>?

where <row>, <modifier>, and <step A> together form a store row address (see section 7.3.3 for details).

Addressing mode

Mode B addressing is used (see section 7.1.2 for details). The effective ADDR field and the effective INT field together define the store row address.

Notes

See notes for the XIC instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
XIQ SPLANE + 2.12 (M4)     ! SET EACH BIT OF THE SELECTED
                           ! STORE ROW TO THE CORRESPONDING BIT OF
                           ! THE Q PLANE, BUT ONLY WHEN THE
                           ! CORRESPONDING BIT OF THE A PLANE IS TRUE
                           !
                           ! THE STORE ROW ADDRESS IS CONSTRUCTED
                           ! USING MODE B ADDRESSING
```

XQ XQ

Function

XQ sets a given store row to the corresponding Q plane row.

Syntax

XQ <row><modifier>?<step A>?

where <row>, <modifier>, and <step A> together form a store row address (see section 7.3.3 for details).

Addressing mode

Mode B addressing is used (see section 7.1.2 for details). The effective ADDR field and the effective INT field together define the store row address.

Notes

See notes for the XAN instruction

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
XQ  18.4                        ! SET ROW 4 OF STORE PLANE 18 TO
                                ! CORRESPONDING ROW OF Q PLANE
```

XR XRN

Function

XR copies a given MCU register or the edge register into a given store row.

XRN copies the inverse of a given MCU register or the edge register into a given store row.

Syntax

XR <MCU-or-edge-register><row><modifier>?<step A>?
XRN <MCU-or-edge-register><row><modifier>?<step A>?

where <row>, <modifier>, and <step A> together form a store row address (see section 7.3.3 for details).

Addressing mode

Mode B addressing is used (see section 7.1.2 for details). The effective ADDR field and the effective INT field together define the store row address.

Notes

1 to 5	See notes for the XAN instruction
6	*MCU-or-edge-register* specifies the value in the MCUR field of the instruction which specifies the register to be copied
7	If an MCU register is specified then its value is extended on the left with zeros if necessary, to make an *ES*-sized value. This value, or its inverse, is written to the row

Possible run-time program errors

A run-time error will occur if the effective ADDR value is outside the range defined by the DAP program block.

Example

```
XR  ME  28.10                ! SET ROW 10 OF STORE PLANE 28 TO
                             ! THE CONTENTS OF THE EDGE REGISTER.
```

Index

There are no references in this index to appendix F. An instruction that is only mentioned in Appendix F has no entry in the index; an instruction that is mentioned in appendix F and elsewhere in the manual does not have an appendix F entry in the index.