MEMORANDUM

Distribution: XAP1 Release

Prepared By: Steve Allen

Subject: Macro Assembler User Manual

XAP1

MACRO ASSEMBLER

USER MANUAL

CONTENTS:

1 INTRODUCTION

The ‘xapasm’ Macro Assembler forms an integral part of the XAP ASIC Processor development toolkit. It can be used as a stand-alone assembler or in the final processing stage of the ‘C’ programming environment; the ANSI ‘C’ compiler generating assembler as its output.

The philosophy behind the XAP has been to create a low-risk environment in which to develop low cost, low power, embedded solutions. For this reason a “keep-it-simple” strategy has been adopted through-out the assembler, allowing the programmer to remain close to the physical hardware model. The more powerful abstract features such as Macros, Conditional Assembly and Expression Evaluation can then be used to aid the readability and understanding of the code, as and when required.

The key features of the Macro Assembler can be summarized as:

- Concatenated assembly of all source files - no linker.
- Microsoft® MASM style macro assembler.
- Powerful expression evaluator.
- Conditional assembly.
- INCLUDE files supported.
- STRUC type supported.
- Large Literals and Branches handled automatically.
- Original line number propagation from ‘C’ compiler.
- Symbol Table and Code files generation for the Simulator / Emulator.

2 USING THE MACRO ASSEMBLER

The XAP Macro Assembler is a two pass assembler and linker. Source files are read sequentially and assembled to form a single executable program. Source files can originate from the user or can be generated by the XAP 'C' compiler.

This section deals with the users' view of the assembler, and covers the following areas:

- Getting Started
- Addressing Modes
- Using Segments
- Creating Programs from Multiple Modules
- Defining Labels and Assembler variables
- Using Structures
- Using Operands and Expressions
- Macros
- Conditional Assembly

General facilities which provide support for linkers are discussed in section 3.

2.1 Getting Started

The assembler is invoked from the command line, with a list of source files and an optional output file name (by default the first source file name is used). The source files are assembled and linked sequentially in a single operation.

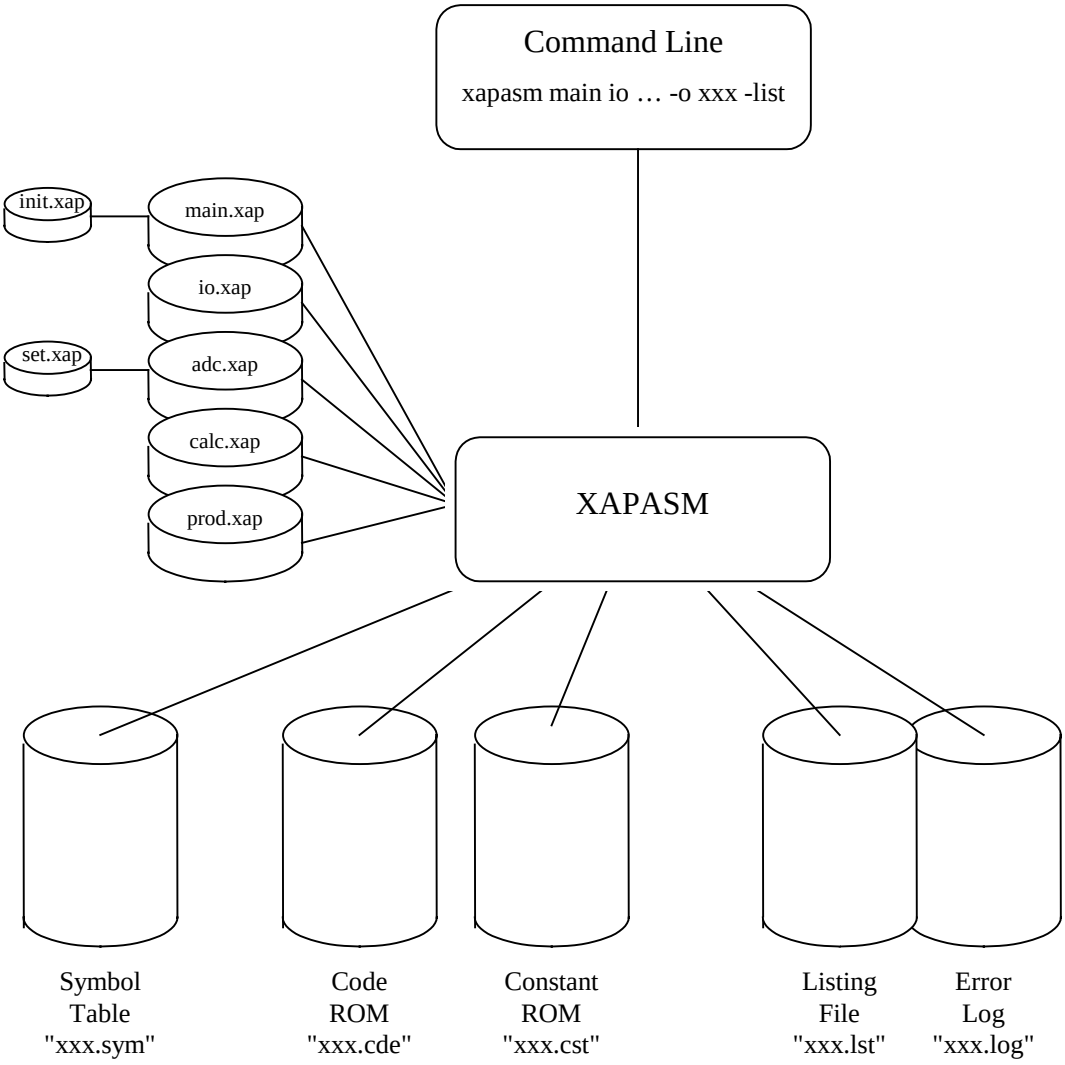
The assembler uses a two pass process, allowing forward references to be made. On the first pass the symbol table is built up (initial substitutions made), and macros and conditional assembly directives expanded. The second pass fills in the forward references and generates the output code. Information is transferred between passes in a working file, which is then deleted at the end of the assembly process.

Assembly continues until all lines have been assembled or an (fatal) error makes further assembly impossible (ie forward reference on a macro expansion).

If no errors are detected three files are generated as output of the assembly process - a symbol table file, a code ROM file and a constant ROM file. The symbol table contains all code and data symbols used by the source files, the code ROM file contains the assembled code, and the constant ROM file contains the values for the constant ROM.
A further two files may be generated during the assembly process - an error log and a listing file. The error log contains a detailed description of any errors detected, and is only generated if errors are found. The listing file contains details of the assembled code along with a summary of equates, labels, modules and errors, and is only generated if the '-list' flag is set.

2.1.1 Assembly Process

The assembly process can be viewed schematically as follows :



Used by the Emulator/Simulator

2.1.2 Invoking the Assembler

The assembler is invoked from the command line with a number of arguments.

xapasm src_file1 [src_file2 ...] [-cmdfile file] [-list] [-supmac] [-supinc] [-quiet] [-o out_file] [-c|s]

Arguments:

src_file1	- first source file.
src_file2 ...	- additional source files.
-cmdfile ‘file’	- read arguments from 'file'.
-quiet	- suppress the screen banner.
-list	- generate list file.
-supmac	- suppress macro expansion in the list file
-supinc	- suppress include file expansion in the list file
-o ‘out_file’	- set the output file name to 'out_file'.
-c	- generate global symbol list. (Linker support only)
-s	- generate segment sizing info. " "

([] indicates an optional argument.)

- If no arguments are given, a command format message is displayed.
- The output file name (‘out_file’) provides the base part of the output file names only. The extensions (.cst, .cde, .err, .lst, .lif) cannot be changed.

2.1.3 Typical Program Structure

A typical program will contain :

- MODULE definition.
- DATA definitions.
- CODE statements.
- ENDMOD statement.

Program - “main.xap”

```
                MODULE   test           ; define MODULE 'test'

                .DATA                   ; select DATA segment
var1            DS    1                 ; var1 at address H'0000
var2            DS    1                 ; var2 at address H'0001

                .CODE                   ; select CODE segment
start:          ld    al,#H'3F
                st    al,@var1          ; store H'3F in var1

                ld    al,#B'100101      ; load al with D'37
                add   al,@var1
                st    al,@var2          ; store result in var2

                ENDMOD   test           ; END program
```

Invoking the assembler with ‘xapasm main -list’ produces the following output:

```
XAP Assembler Vx.xx  (xx/xx/xx)
(c) Cambridge Consultants Ltd 1998
NO ERRORS FOUND
```

Code and Constant Table files are produced in a format compatible with HILO ROM files :

```
** <Header>
WORDSIZE xx
BASE HEX : HEX
<Address> : <Code/Data> ** <Source line>
END ROM
```

(** = HILO comment character)

Code Table - “main.cde”

```
** TITLE        : "Code Table"
** DATE         : Tue May 19 13:19:36 1998
** COMMAND LINE : xapasm main -list
**
**
** ROM START ADDRESS 0000
WORDSIZE 18
BASE HEX : HEX
0000 : 03F14 ** start:              ld         al,#H'3F
0001 : 00025 **                     st         al,@var1
0002 : 02514 **                     ld         al,#B'100101
0003 : 00035 **                     add        al,@var1
0004 : 00125 **                     st         al,@var2
END ROM
```

Constant Table - “main.cst”

```
** TITLE        : "Constant Table"
** DATE         : Tue May 19 13:19:36 1998
** COMMAND LINE : xapasm main -list
**
**
** ROM START ADDRESS 0000
WORDSIZE 16
BASE HEX : HEX
END ROM
```

The '** ROM START ADDRESS' line is a special comment used by the Simulator / Emulator to identify the start address of the ROM (see section 2.3.5). The addresses in the body of the file are defined relative to the start address.

A Symbol Table is produced for use by the symbolic debugger in the Simulator/Emulator, and contains a list of all CODE and DATA symbols along with their associated addresses.

CODE symbols are automatically generated by the assembler for each line of source code that has real code associated with it. This allows the symbolic debugger to access the original source code line.

Symbol Table - "main.sym"

```
** TITLE        : "Symbol Table"
** DATE         : Tue May 19 13:19:36 1998
** COMMAND LINE : xapasm main -list
**
**
.CODE
test.start, 0000
test.'main.xap'.?8.1, 0000
test.'main.xap'.?9.1, 0001
test.'main.xap'.?11.1, 0002
test.'main.xap'.?12.1, 0003
test.'main.xap'.?13.1, 0004
eof_label.'main.xap'.?999999.1, 0005
.DATA
test.var1, 0000
test.var2, 0001
.END
```

By setting the “-list” option on the command line a listing file is produced in the following format:

```
<Header>
<Line Num> <Address> <Segment> <Data> : <Source line>
<Summary>
<Error Summary>
```

where segments have the following codes:

:	D	- data segments (.DATA, .VAR)
	R	- ROM constant segment (.CONST, .LITVALS)
	U	- user-defined segments
	C	- code segment (.CODE)

Listing - “main.lst”

```
################################################################################
#                                                                              #
# XAP Assembler - Vx.xx  (xx/xx/xx)               Tue May 19 13:19:36 1998  #
#                                                                              #
# COMMAND LINE : xapasm main -list                                             #
#                                                                              #
#                                                                              #
#                                                                              #
# (c) Cambridge Consultants Ltd 1998                                           #
#                                                                              #
################################################################################


File: main.xap

    1              :                      MODULE     test                           ; define MODULE 'test'
    2              :
    3              :                      .DATA                                     ; select DATA segment
    4 0000 D  ???? : var1                 DS         1                              ; var1 at address H'0000
    5 0001 D  ???? : var2                 DS         1                              ; var2 at address H'0001
    6              :
    7              :                      .CODE                                     ; select CODE segment
    8 0000 C 03F14 : start:               ld         al,#H'3F
    9 0001 C 00025 :                      st         al,@var1                       ; store H'3F in var1
   10              :
   11 0002 C 02514 :                      ld         al,#B'100101                   ; load al with D'37
   12 0003 C 00035 :                      add        al,@var1
   13 0004 C 00125 :                      st         al,@var2                       ; store result in var2
   14              :
   15              :                      ENDMOD     test                           ; END program


SUMMARY
-------

MODULES : 1

test                            : Start Add = H'0000, Size = H'0005

LITVALS : 0

CODE SYMBOLS : 1

test.start                      : Address = H'0000

DATA SYMBOLS : 2

test.var1                       : Address = H'0000
test.var2                       : Address = H'0001

DATA STRUCTURE SYMBOLS : 0

EQUATE SYMBOLS : 0

REDEFINABLE EQUATE SYMBOLS : 0

STRING EQUATE SYMBOLS : 0

SEGMENTS : 4

$??CODE                         : Segment Start Add = H'0000, Size = H'0005
$??DATA                         : Segment Start Add = H'0000, Size = H'0002
$??VAR                          : Segment Not Used
$??CONST                        : Segment Not Used



ASSEMBLY
--------

NO ERRORS FOUND
NO WARNINGS
```

Other Codes

When a source line produces no ROM output, the listing file has the following format:

```
<Line Num> <List Code> : <Source line>
```

where:

<List Code>: X - source line not processed due to conditional assembly.
M - macro definition.
S - struc definition
– - non output producing directive.

Large Literals

Large literal substitutions (see section 2.2.4) are indicated in the listing file by adding the automatically generated litval variable name ($??xxxxL) to the end of the source line concerned. The values assigned to the litval variables are included in the listing file as pseudo code appended to the end of the source code.

```
   1              :                     MODULE
   2              :                     .LITVALS   H'0000
   3              :
   4              :                     .CODE
   5 0000 C 00015 :                     ld         al, #NOT H'800       $??0000L
   6 0001 C 00115 :                     ld         al, #NOT H'7FFF      $??0001L
   7 0002 C 00215 :                     ld         al, #LWRD(NOT H'8000) $??0002L
   8              :
   9              :                     ENDMOD
   .              :                     * .SEG     LITVALS
   .              :                     * ORG      H'0000
   . 0000 R  F7FF : $??0000L            * DC       H'F7FF
   . 0001 R  8000 : $??0001L            * DC       H'8000
   . 0002 R  7FFF : $??0002L            * DC       H'7FFF
```

Macro Expansion

Macro expansions are indented, and high-lighted with a “*”, in the listing file.

```
   1              :                     MODULE
   2              :
   3 M            : first               MACRO
   4 M            :                     ld         x, #23
   5 M            :                     second                                  ; nested MACRO
   6 M            :                     ENDMAC
   7              :
   8 M            : second              MACRO
   9 M            :                     st         x, @H'02
  10 M            :                     ENDMAC
  11              :
  12              :                     .CODE
  13              :                     first                                   ; call MACRO first
   . 0000 C 01718 :                     * ld       x, #23
   .              :                     * second                                ; nested MACRO
   . 0001 C 00229 :                     * st       x, @H'02
   .              :                     * ENDMAC
   .              :                     * ENDMAC
  14              :                     ENDMOD
```

2.2 Address Modes

2.2.1 Address Modes

The XAP supports the following address modes :

- Immediate - load/store immediate data
- Direct - load/store/[branch to] the contents of address (arg)
- Indexed X - load/store the contents of address (arg+ X)
- Indexed Y - load/store the contents of address (arg+ Y)
- Relative PC - branch to address (arg + PC)
- Relative X - branch to address (arg + X)

The Macro Assembler encodes these address modes in the following formats:

2.2.2 General Instruction Format

Name	Assembler Syntax	'Data' (Source / Destination)	Valid range of arg
Immediate	#arg	Argument	See Large Literals (below)
Direct	@arg	Contents of address (arg)	H'0..H'1FF, H'FE00..H'FFFF
Indexed X	@(arg, X)	Contents of address (arg + X reg)	-512..511, H'FE00..H'FFFF
Indexed Y	@(arg, Y)	Contents of address (arg + Y reg)	-512..511, H'FE00..H'FFFF

2.2.3 Branch Instruction Format

Name	Assembler Syntax	'Branch Address' (new Program Counter value)	Valid range of arg
Relative PC	arg	PC + Argument	See Large Literals (below)
Direct	@arg	Contents of address (arg)	H'0..H'1FF, H'FE00..H'FFFF
Relative X	arg, X	Argument + X reg	-512..511, H'FE00..H'FFFF
Indexed Y	@(arg, Y)	Contents of address (arg + Y reg)	-512..511, H'FE00..H'FFFF

- @ indicates 'contents of'.
- # indicates 'immediate'.
- SP (Stack Pointer) can be used as an alias for Y.
- The program counter (PC) is not pre-incremented.

2.2.4 Large Literals

The XAP instruction set provides a 10 bit field (sign extended to 16 bits) for the argument (or literal) of an instruction. This gives a value of -512..511 or H'FE00..H'FFFF in the Immediate address mode, or a branch address of ±512 in the Relative PC address mode. When values outside this 10 bit range (known as large literals) are required, they must first be stored as constants in the Direct address space (H'0000..H'01FF and H'FE00..H'FFFF), and then accessed using the Direct address mode.

The assembler can carry out the translation from Immediate or Relative PC addressing modes to Direct addressing automatically using a special LITVALS segment (see section 2.3.2). The automatic resolution of large literals is enabled simply by defining the LITVALS segment. Whenever the assembler detects a literal outside the -512..511 range the number is stored in the LITVALS segment and its address substituted in the instruction. Neither the code size or execution speed is effected by the change of addressing mode. The LITVALS segment must be defined to lie within the Direct address space.

Program - "address.xap"

```
                MODULE   address_example

                .DATA                   ; select DATA segment
                ORG   H'FE00            ; set DATA location counter to H'FE00
fred            DS    3                 ; reserve 3 words and label address fred

                ORG   H'FF00            ; set DATA location counter to H'FF00
stack_space     DS    H'100             ; reserve 256 words for the stack

                .CONST                  ; select CONST segment
                ORG   H'0000            ; set CONST location counter to H'0000
v1              DC    H'3002            ; define a constant v1 with value H'3002

JIM             EQU   H'1A0             ; give JIM the value H'1A0
bill            EQU   <(2,sp)>          ; bill equals the text string '(2,SP)'
stack           =     H'FF00            ; assign H'FF00 to variable stack

                .CODE                   ; select CODE segment
                ld    al,#JIM           ; al = H'1A0
                st    al,@fred          ; fred[0] = al
                ld    x,#1              ; x = 1
bob:            ld    ah,@(fred,x)      ; ah = fred[1]
                ld    sp,#stack         ; sp = H'FF00
                st    ah,@bill          ; store AH at addr H'102
                bne   bob               ; branch if not equal to label bob
                bpl   @v1               ; branch if plus to H'3002
                bra   $-5               ; branch to bob - ($ = location counter)

                ORG   H'3002
here:           sleep

                ENDMOD   address_example
```

2.3 Segments

2.3.1 Pre-Defined Segments

The programmer has access to four pre-defined segments, allowing common data types to be grouped together. These segments can be selected with the following directives:

Directive	Segment	Use
.CONST	Constant	ROM Constants stored in Direct Address space (H'0..H'1FF and H'FE00..H'FFFF)
.DATA	Data	IO and large tables of constants / variables (stored outside Direct Address space)
.VAR	Variable	RAM Variables stored in Direct Address space
.CODE	Code	Program code.

The DATA, VAR and CONST segments all map on to the same physical address space (data space). The CODE segment has its own physical address space (code space) in the XAP's Harvard architecture. Checks are made to ensure that the same physical addresses are not used in different segments.

A segment's current position within the physical address space is stored in a special assembler variable called a location counter. Each segment has its own location counter, which is incremented automatically when an item is added to the segment. Items are written to segments using DS and DC directives. The CODE segment can also be written to using special XAP instruction statements (add, ld, mult, etc).

Segment location counters, initialised to address H'0000, are stored in the global variables $??CODE, $??DATA, $??VAR and $??CONST. Reading these variables returns the current address in the segment, which can then be used within expressions to calculate sizes of arrays / strings etc.

At any one time only the most recently selected segment is active. All DS, DC and code statements (instructions) are stored in this segment as they are processed. The location counter for the current segment can be accessed using the location counter ($) global variable.

It is possible to change the current address in the currently selected segment with the ORG directive - the location counter is assigned the new address. The address can be changed at any point after the segment has been selected, and as many times as required. By reassigning the current address it is possible to accommodate gaps in the address space or place tables at specific locations.

```
ORG   address
```

Restrictions are applied to the CODE segment to ensure that addresses can only ever increment. This guarantees that the order in which the code is assembled will always match the order in which the source files appear on the command line. No restrictions apply to the DATA, VAR and CONST segments, which can be freely navigated.

The location counter variables are special assembler variables and can only be reassigned with the ORG directive.

2.3.2 Litvals Segment

A fifth pre-defined segment (LITVALS) is available only to the assembler; this also maps into the data space, sharing physical addresses with the DATA, VAR and CONST segments. The LITVALS segment is used by the assembler to store direct-access-automatic literals. These are generated when the assembler encounters a large literal (immediate data and branch addresses outside the 10 bit instruction argument range of -512..511 and H'FE00..H'FFFF). The instruction's address mode is changed to direct, pointing to the literal now stored in the direct address space.

In order to handle these large literals the LITVALS segment start address must be defined using:

```
.LITVALS   address     ; start of the automatic literals segment
```

If the LITVALS segment is not defined, large literals will cause an assembly error.

2.3.3 User Segments

In addition to the pre-defined segments, any number of user-defined segments can also be created. They share the same physical address space (data space) as the DATA, VAR, CONST, and LITVALS segments, but maintain their own location counters and can be used to hold large arrays, pre-defined variables, tables etc. The segments are defined, and selected, using the .SEG directive; the first time the segment is encountered it is defined, subsequently it is simply selected.

```
.SEG LOOKUP_TABLE
```

The current value of the segment's location counter is stored in the global variable $??*segment_name.*

The .SEG directive can also be used to select the pre-defined CODE, DATA, VAR, CONST segments.

2.3.4 Location Counter

The location counter ($) is a special symbol that, during assembly, represents the address of the statement currently being assembled.

```
            .CONST
string      DC    P'Who wants to count every word in a string'
            DC    P'especially if it might be changed later'
lstring     DC    $ - string
```

2.3.5 ROM Start Address

ROM start addresses are stored in the header section of the code and constant table files. They are read by the Simulator / Emulator and used as an offset for the addresses in the rest of the file. HILO ignores the start address information, assuming that the address decode logic has been defined correctly.

The code segment ROM start address is taken as the first address in the CODE segment to contain code. Code normally starts at H'0000, however this can be changed using either the ORG directive or by reserving an area of ROM (from H'0000) with the DS directive. A non-zero start address will generate a warning.

The constant segment ROM start address is taken as the first address for which a constant is defined (DC). A non-zero start address will generate a warning.

Program - “segment.xap”

```
               .LITVALS H’100

               MODULE   segment_example

               .DATA                    ; selects DATA segment
               ORG   H'FF00             ; set DATA location counter to H'FF00
fred           DS    1                  ; reserve 1 word and label address fred

               .VAR
               ORG   H'FE80             ; set VAR location counter to H'FE80
io_space       DS    H'30               ; reserve space for IO


; Note : the first use of the '.SEG' directive creates the segment
; subsequent uses select the same segment

               .SEG  INIT_STR           ; create the INIT_STR user-defined segment
               ORG   H'FE00             ; set INIT_STR location counter to H'FE00
INIT_ADDR      =     $??INIT_STR        ; INIT_ADDR = start of INIT_STR segment
time           DS    1
cost           DS    1
LEN            =     $ - INIT_ADDR      ; LEN = the length of INIT_STR segment

; Note : $ and $??INIT_STR have the same value within the segment and can
; be interchanged

               .CONST                   ; select CONST segment
               ORG   H'0020             ; set CONST location counter to H’0020
v1             DC    H'3002             ; define a constant v1 with value H’3002

               .SEG  INIT_STR           ; select the INIT_STR user-defined segment
result         DS    1
LEN            =     $ - INIT_ADDR      ; LEN = the length of INIT_STR segment

               .CODE                    ; select CODE segment
               ld    x,#H'1234          ; generates a LITVAL as H'1234 is > 10bits
               st    x,@fred            ; store x at H'FF00

               .CONST                   ; re-select CONST segment, = H'0021
v2             DC    'Hello World'      ; store the string 'Hello World'


               .CODE                    ; re-select CODE segment, = H'0002
               add   x,@(io_space+6)    ; add the contents of H'FE86 to x

               ENDMOD   segment_example
```

Listing - “segment.lst”

```
################################################################################
#                                                                              #
# XAP Assembler - Vx.xx  (xx/xx/xx)                Tue May 19 12:44:04 1998  #
#                                                                              #
# COMMAND LINE : xapasm segment -list                                          #
#                                                                              #
#                                                                              #
#                                                                              #
# (c) Cambridge Consultants Ltd 1998                                           #
#                                                                              #
################################################################################



File: segment.xap

    1              :                      .LITVALS   H'100
    2              :
    3              :                      MODULE     segment_example
    4              :
    5              :                      .DATA                                ; selects DATA segment
    6              :                      ORG        H'FF00                    ; set DATA location counter to H'FF00
    7 FF00 D  ???? : fred                 DS         1                         ; reserve 1 word and label address fred
    8              :
    9              :                      .VAR
   10              :                      ORG        H'FE80                    ; set VAR location counter to H'FE80
   11 FE80 D  ???? : io_space            DS         H'30                      ; reserve space for IO
   12              :
   13              :
   14              : ; Note : the first use of the '.SEG' directive creates the segment
   15              : ; subsequent uses select the same segment
   16              :
   17              :                      .SEG       INIT_STR                  ; create the INIT_STR user-defined segment
   18              :                      ORG        H'FE00                    ; set INIT_STR location counter to H'FE00
   19              : INIT_ADDR            =          $??INIT_STR               ; INIT_ADDR = start of INIT_STR segment
   20 FE00 U  ???? : time                 DS         1
   21 FE01 U  ???? : cost                 DS         1
   22              : LEN                  =          $ - INIT_ADDR             ; LEN = the length of INIT_STR segment
   23              :
   24              : ; Note : $ and $??INIT_STR have the same value within the segment and can
   25              : ; be interchanged
   26              :
   27              :                      .CONST                               ; select CONST segment
   28              :                      ORG        H'0020                    ; set CONST location counter to H*0020
   29 0020 R  3002 : v1                   DC         H'3002                    ; define a constant v1 with value H*3002
   30              :
   31              :                      .SEG       INIT_STR                  ; select the INIT_STR user-defined segment
   32 FE02 U  ???? : result               DS         1
   33              : LEN                  =          $ - INIT_ADDR             ; LEN = the length of INIT_STR segment
   34              :
   35              :                      .CODE                                ; select CODE segment
   36 0000 C 10019 :                      ld         x,#H'1234          $??0000L   ; generates a LITVAL as H'1234 is > 10bits
   37 0001 C 30029 :                      st         x,@fred                   ; store x at H'FF00
   38              :
   39              :                      .CONST                               ; re-select CONST segment, = H'0021
   40 0021 R  0048 : v2                   DC         'Hello World'             ; store the string 'Hello World'
    . 0022 R  0065 :
    . 0023 R  006C :
    . 0024 R  006C :
    . 0025 R  006F :
    . 0026 R  0020 :
    . 0027 R  0057 :
    . 0028 R  006F :
    . 0029 R  0072 :
    . 002A R  006C :
    . 002B R  0064 :
   41              :
   42              :
   43              :                      .CODE                                ; re-select CODE segment, = H'0002
   44 0002 C 28639 :                      add        x,@(io_space+6)           ; add the contents of H'FE86 to x
   45              :
   46              :                      ENDMOD     segment_example
    .              :                      * .SEG     LITVALS
    .              :                      * ORG      H'0100
    . 0100 R  1234 : $??0000L             * DC       H'1234


SUMMARY
-------

MODULES : 1

segment_example                : Start Add = H'0000, Size = H'0003

LITVALS : 1

$??0000L                       : Address = H'0100, Value = H'1234

CODE SYMBOLS : 0
```

Listing - “segment.lst” (continued)

```
DATA SYMBOLS : 7

segment_example.fred             : Address = H'FF00
segment_example.io_space         : Address = H'FE80
segment_example.time             : Address = H'FE00
segment_example.cost             : Address = H'FE01
segment_example.v1               : Address = H'0020
segment_example.result           : Address = H'FE02
segment_example.v2               : Address = H'0021

DATA STRUCTURE SYMBOLS : 0

EQUATE SYMBOLS : 0

REDEFINABLE EQUATE SYMBOLS : 2

segment_example.INIT_ADDR        : Value = H'0000FE00, D'65024
segment_example.LEN              : Value = H'00000003, D'3

STRING EQUATE SYMBOLS : 0

SEGMENTS : 6

$??CODE                          : Segment Start Add = H'0000, Size = H'0101
$??DATA                          : Segment Start Add = H'FF00, Size = H'0001
$??VAR                           : Segment Start Add = H'FE80, Size = H'0030
$??CONST                         : Segment Start Add = H'0020, Size = H'000C
$??LITVALS                       : Segment Start Add = H'0100, Size = H'0001
$??INIT_STR                      : Segment Start Add = H'FE00, Size = H'0003



ASSEMBLY
--------

NO ERRORS FOUND
WARNINGS : 1

Warning     : Non-zero ROM Start Address
Description : Constant ROM Start Address = H'0020
```

2.4 Modules

Most assembly-language programs are created from several modules. When several modules are used, the scope of symbols becomes important.

Unlike conventional programming languages, separation into files does not define or modify the modular structure. Modules are the only divider, source files are simply concatenated. Files can contain any number of modules.

2.4.1 Defining Modules

Modules are surrounded by the MODULE and ENDMOD directives. The file name is used as the module name if no name is specified with the MODULE directive. Obviously if a file contains several modules, only one can be named in this way. The ENDMOD directive can have an optional module name, matching the MODULE directive.

```
MODULE      [name]
ENDMOD      [name]
```

Symbols declared in INCLUDE files (see section 2.13.2) within a module remain local to that module; modules are the only divider, not files. A module may contain any number of INCLUDE files.

```
              MODULE   main
              .LITVALS H'0100
              .CONST
              ORG   H'0000
mess          DC    'Hello',0         ; local to module main
temp          DC    1                 ; local to module main
              .VAR
              ORG   H'FE00
$x            DS    1                 ; global variable
$ptr          =     2                 ; define global assembler variable
              .CODE
              bra   $calc
$return:      sleep                   ; global label
              ENDMOD   main

              MODULE   sort
              .CONST
temp          DC    H'0F01            ; local to module sort
              .CODE
$calc:        ld    ah,@temp          ; ah = H'0F01
              st    ah,@$x            ; store AH in $x (declared in main)
              ld    al,#$ptr          ; al = 2
              ld    ah,@^main.temp    ; access temp in main with '^' prefix
              bra   $return
              ENDMOD   sort
```

2.5 Symbols

2.5.1 Local Symbols

All labels, assembler variables, structures and structure type definitions are declared local to a module, unless prefixed by the special global identifier ($).

Symbols local to another module can be accessed by prefixing the symbol name with '^*module*.', ie '^main.temp'

2.5.2 Global Symbols

Global symbols are symbols prefixed by a '$'. They are accessible from any module in any file.

Macros are always treated as global.

2.5.3 Symbol Names

Symbol names can be of any length and :

- Start with 'A..Z', 'a..z', '_' , '?' or '$'.
- Contain only the following characters: 'A..Z', 'a..z', '0..9', '_' or '?'
- Symbol names conform to the following templates :-

$??name	- segment location counters and general assembler variables for global symbols which must not interfere with the 'C' name space.	
$??hex	- MACRO label.	(reserved)
$??hexL	- litval.	(reserved)
$?name	- 'C' automatically generated global symbol.	(reserved)
?name	- 'C' automatically generated local symbol.	(reserved)
$name	- global variable / label.	
name	- local variable / label.	

- Structure fields can be accessed as a special type of symbol where name in the above is of the form structure_name.field_name. See section 2.10.
- Names are case sensitive.

- Accesses to local symbols in different modules use the following format:

  ^module.name - local symbol access

- Original line number labels have the following format:

  ^module.'file_name'.?num.num - original line number label

- Symbols must not match an OPERATOR or REGISTER name. The comparison is made in a case insensitive way.

2.5.4 Symbol Classes

The assembler distinguishes between three different symbol classes:

- Labels, Assembler variables and Structures
- Structure Types and MACROs
- Structure Fields

Each symbol class has its own set of "Operators", "Reserved Words" and "Name Space" (set of symbol names). Structure fields have a separate "Name Space" for each structure type.

It is possible for a structure, structure type and structure field all to have the same name, as they belong to different "Name Spaces". Structure fields belonging to different structure types can also have the same name.

2.6 Labels and Assembler Variables

2.6.1 Labels

Labels give symbolic names to the addresses of instructions or data. These labels can be used as the operands to branch instructions, or as addresses in expressions. Labels can be created in both the code and data segments.

name:

```
                .CONST
table:                                  ; label (in data segment)
v1              DC      H'1000          ; label
v2              DC      H'10F3          ; label

                .CODE
                ld      x,#table        ; first element of table

                cmp     al,#5
                bne     not_equal       ; branch if al not equal to 5
                bra     equal           ; branch always

not_equal:      ...                     ; label (in code segment)
                ...
                bra     done

equal:          ...                     ; label
                ...
                bra     done
```

Labels can also be created using the LABEL directive, or by preceding the DC (Define Constant) or DS (Define Storage) directive with a name.

```
name  LABEL
name  DC …
name  DS …
```

2.6.2 Assembler Variables

Assembler variables are used to store text strings or numeric values. There are three basic variable types:

- Numeric equates
- String equates
- Redefinable equates

Global variables are very useful for passing information between modules / files and to external linkers (see -c assembler option; Section 3). All variables are assigned at assembly time.

Numeric Equates

Numeric equates assign a numeric constant to a variable. Variables automatically record the type (integer or fractional) of the numeric constant they hold. The assembler replaces each occurrence of name with the value of expression, wherever it occurs within scope, whether before or after the definition.

```
name  EQU   expression
```

Once a numeric equate has been defined with the EQU directive, it cannot be redefined within its scope (module or global).

No storage is allocated for the symbol.

```
column      EQU   80                  ; numeric constant 80
row         EQU   25                  ; numeric constant 25
screenful   EQU   column * row        ; numeric constant 2000
line        EQU   row                 ; same as "row", numeric constant 25
$screensize EQU   screenful           ; screen size in bytes (global variable)

forward     DC    YES                 ; use before definition
YES         EQU   1
```

String Equates

String equates are used to assign a text string constant to an assembler variable. String equates can be used in a variety of contexts, including defining aliases and string constants.

```
name  EQU   < string >
```

The assembler replaces each occurrence of name with string, wherever it occurs within scope, whether before or after the definition. If the string matches the name of a variable, it creates an alias for the variable.

Like numeric equates, the value of the variable cannot be redefined within its scope (module or global).

```
pi          EQU   <3.1415>              ; string constant for "3.1415"
prompt      EQU   <'Type Name : '>      ; string const "'Type Name : '"
arg1        EQU   <(0, SP)>             ; string constant for "(0, SP)"
arg2        EQU   <pi>                  ; alias of "pi", arg2 replaced with 3.1415

            .CONST
message     DC    prompt                ; expands to DC 'Type Name: '

            .CODE
            ld    al,@arg1              ; expands to ld al,@(0, SP)
```

Redefinable Equates

Redefinable equates assign a value to an assembler variable, that may be redefined at a later time. Variables automatically record the type (integer or fractional) of the value they hold. The variable can be redefined at any point during assembly.

```
name  = expression
```

Unlike numeric and string equates, variables defined using redefinable equates can only be used in statements following their definition.

```
errors      =     errors + io_errors   ; keep track of the number of errors
                                       ; in error table.

            IF errors GT error_table_size
               .ERR Too many errors    ; stop assembly if too many errors
            ENDIF

$NUM_FILES  =    $NUM_FILES + 1        ; global number of files variable
```

2.7 Numeric Data Types

The Macro Assembler supports both Integer and Floating Point arithmetic.

2.7.1 Floating Point Arithmetic

Floating point values can be freely used in expressions and assembler variables. There is, however, no XAP instruction set support for floating point arithmetic.

Floating point arithmetic can be handled efficiently by restricting the number range to $-1.0 \leq x < 1.0$, and using the fixed point fractional arithmetic support of the XAP. Floating point numbers in the range $-1.0 \leq x < 1.0$ are automatically converted to fractionals when assigned to memory (see section 2.7.2).

Numbers outside the range $-1.0 \leq x < 1.0$ are converted to integers when assigned to memory, with a range violation warning. They can be explicitly converted to the nearest whole integer using the INT operator (see section 2.9.7).

Full floating point arithmetic can be implemented by writing specialist ‘Math Libraries’. This will lead to a considerable increase in code size and design complexity.

2.7.2 Fractional Arithmetic

A fractional is a fixed point (two’s complement number) expressing values in the range $-1.0 \leq x < 1.0$. Floating point numbers in the range $-1.0 \leq x < 1.0$ are automatically converted to fractionals when they are assigned to memory. Numbers outside this range are converted to integers and a range violation warning issued (unless explicitly assigned with the INT operator).

Fractionals have the following bit representation:

Most Significant Bit				Least Significant Bit
-1	2^{-1}	2^{-2}	2^{-3}	…

0.5	=	H’40000000	(32-bit)	H’4000	(16-bit)
-1.0	=	H’80000000	(32-bit)	H’8000	(16-bit)
-0.5	=	H’C0000000	(32-bit)	H’C000	(16-bit)
-0.183	=	H’E872B021	(32-bit)	H’E873	(16-bit)
0.206	=	H’1A5E353F	(32-bit)	H’1A5E	(16-bit)

By comparison, integers have the following bit representation:

Most Significant Bit				Least Significant Bit
-2^{31} or -2^{15}	…	2^2	2^1	2^0

The XAP’s instruction set has been designed to support both fractional and integer arithmetic. Addition and Subtraction operations work on signed or unsigned, integer or fractional values. Multiplication and Division operations differ slightly between fractional and integer (see the MULT and DIV instructions in section 4).

2.7.3 Long Integers and Long Fractionals

Both integer and fractional arithmetic can be extended to 32-bits using the carry / borrow flag. The Macro Assembler provides the operators HWRD and LWRD, to select the high and low words of 32-bit expressions, allowing easy assignment of 32-bit constants.

2.7.4 Allowed Number Ranges

Assembler variables and expressions are evaluated and stored as 32-bits numbers. Numbers assigned to memory locations are stored in 16-bits, and numbers used as instruction operands are stored in 10-bits.

Error checking is performed to ensure that the number remains within range.

- *Expressions and Assembler Variables (signed 32-bit)*

  Integer range: $-2147483648 \leq x \leq 2147483647$
  Floating Point range: $\pm 10^{-38}$ to 10^{+37}

- *Addresses/Constants assigned to Memory (16-bit)*

  Integer range (signed/unsigned): $-32768 \leq x \leq 65536$
  Fractional range (signed): $-1.0 \leq x < 1.0$

- *Operands (10-Bit) - see section 2.2*

Immediate:	-512 .. 511,	H'FE00 .. H'FFFF
Direct Address:	0 .. H'1FF,	H'FE00 .. H'FFFF
Indexed X,Y:	-512 .. 511,	H'FE00 .. H'FFFF
Relative X, PC:	-512 .. 511,	H'FE00 .. H'FFFF

2.8 Defining and Initializing Data

The DC (Define Constant) directive is used to declare ROM space. The DS (Define Storage) directive is used to reserve RAM space. RAM space cannot be initialised.

```
[label_name]  DC expression  (literal values)
[label_name]  DS expression  (number of words to reserve)
```

```
              .CONST
v1            DC    'HELLO',0         ; null terminated text string - 6 words
v2            DC    "HELLO"           ; null terminated text string - 6 words
v3            DC    P'Hello\0'        ; packed null terminated string - 3 words
v4            DC    'A','B'           ; ASCII constants
v5            DC    H'10A0            ; hex
v6            DC    D'35              ; decimal - default
v7            DC    B'01100101        ; binary
v8            DC    0.317             ; floating point

              .VAR
v9            DS    (3+5) MOD 3       ; reserve 2 words
```

2.8.1 String Formats

The assembler accepts the following string formats:

Single Quotes	‘Hello’	; standard, single character per word.
Double Quotes	“Hello”	; null terminated.
Packed	P’Hello’	; two characters per word.

Within a string special characters can be inserted using the Back-Slash escape character.

\n	Newline
\r	Carriage Return
\0	Null
\\	Back-Slash
\'	Single Quote
\"	Double Quote
\x*XX*	Hex Code *XX*

2.8.2 Number Formats

Numbers can be entered as Hex, Decimal, Binary or Floating Point as follows:

D’23	Decimal
23	Decimal (default)
H’12F3	Hex
B’1001	Binary
.423	Floating Point / Fractional
‘a’	ASCII

Numbers can also be entered as the result of an expression.

23 E -2	Floating point
6 MOD 2	Decimal

2.8.3 Segment Usage

Although the DS and DC directives can be used in all segments their use should be restricted to the following:

Segment	DC	DS
CODE	Insert an 18 bit instruction	Skip a range of addresses
DATA	Define a table (fixed array)	Define IO space or array
VAR	Not used	Define a variable
CONST	Define a constant / string	Not used
User Defined	No Restriction	No Restriction

2.8.4 Forward References

The XAP Macro Assembler is a two pass assembler. Structures, macros and conditional assembly directives are expanded during pass 1. There must be no remaining forward references after pass 1. Pass 2 is used to assign those variables forward referenced during pass 1 and assemble the actual code. Any expressions unresolvable in pass 2, due to two stage forward referencing, will generate an error message.

```
            .CONST
length      DC    end - hello          ; illegal as double forward referencing
table0      DC    start_pt             ; ok as start_pt has a size of 1
table1      DC    fred

hello       EQU   start_pt
fred        EQU   <'HELLO'>            ; gives error as fred assumed to be size 1
                                       ; as used in a DC forward reference

            .CODE
start_pt:   ld    x,#4
            IF (start_pt EQ hello)     ; illegal as hello not defined on pass 1
               ld    y,#6
            ENDIF
            IF (start_pt LT $)         ; ok as both start_pt and $ are known
               ld    al,#7
            ENDIF
end:
```

Symbols forward referenced with the DC directive are assumed to be of size one. If the symbol is subsequently defined as a type with a size greater than one, an error message will be produced.

Equates embedded within a structure (see section 2.10.4) cannot contain forward references as structures are processed during pass 1 of the assembly process.

2.9 Operands, Operators and Expressions

2.9.1 Operands

Operands are the arguments of instructions, directives or expressions. An operand can take the form of a constant, a label, an assembler variable, a structure, a structure type, a register or an expressions.

The number, type and position of the operand is determined by the type of directive, instruction or operator.

2.9.2 Operators

The assembler provides a variety of operators for combining, comparing, or changing the values of operands in an expression.

- Arithmetic Operators (integer and floating point)
- Bitwise Operators (integer)
- Relation Operators (integer and floating point)
- Logic Operators (integer)
- Assignment Operators

Mixed Expressions

If an Arithmetic or Relation (integer and floating point) operator is presented with one integer and one floating point operand, the result will be a floating point number.

```
pattern = 3 * 1.2                ; pattern = 3.6
```

If a Bitwise or Logic (integer only) operator is presented with a floating point operand in the range $-1.0 \leq x < 1.0$, the operator is applied to the fractional bit representation of the floating point number (see section 2.7.2)

```
pattern = 0.206 AND H'FF00FFF0     ; pattern = H'1A003530
```

If a Bitwise or Logic (integer only) operator is presented with a floating point operand outside the range $-1.0 \leq x < 1.0$, the floating point number is converted to an integer and a range violation warning generated.

```
pattern = 500.2 AND H'FF00FFF0     ; pattern = H'000001F0
```

2.9.3 Arithmetic Operators

These operators perform arithmetic operations on the operands.

Operator	Syntax	Meaning
+	+ *exp*	Positive (unary)
-	- *exp*	Negative (unary)
*	*exp1* * *exp2*	Multiplication
/	*exp1* / *exp2*	Division
MOD	*exp1* MOD *exp2*	Remainder (integer)
ABS	ABS *exp*	Absolute value
+	*exp1* + *exp2*	Addition
-	*exp1* - *exp2*	Subtraction
FEXP	FEXP *exp*	Extract Exponent (dec)
FMAN	FMAN *exp*	Extract Mantissa (dec)
E	*exp1* E *exp2*	*exp1* x 10^*exp2*
FREXP	FREXP *exp*	Extract Exponent (bin)
FRMAN	FRMAN *exp*	Extract Mantissa (bin)
LDEXP	*exp1* LDEXP *exp2*	*exp1* x 2^*exp2*

```
intgr           =     14 * 3            ; 14 * 3 = 42
intgr           =     intgr / 4         ; 42 / 4 = 10
intgr           =     intgr MOD 4       ; 10 mod 4 = 2
intgr           =     intgr + 4         ; 2 + 4 = 6
intgr           =     intgr - 3         ; 6 - 3 = 3
intgr           =     -intgr - 8        ; -3 - 8 = -11
intgr           =     -intgr - intgr    ; 11 - -11 = 22

fm              =     FMAN 23.4         ; fm = .234
fe              =     FEXP 23.4         ; fe = 2
f               =     fm E fe           ; f = 23.4

frm             =     FRMAN 23.4        ; frm = 0.73125
fre             =     FREXP 23.4        ; fre = 5
fr              =     frm LDEXP fre     ; fr = 23.4


                ORG   H'100
a               DS    1
b               DS    1
mem1            EQU   a + 5             ; mem1 = H'105
mem2            EQU   a - 5             ; mem2 = H'FB
```

2.9.4 Bitwise Operators

These operators perform bit manipulation operations on the operands.

Operator	Syntax	Meaning	
SHL	exp SHL count	Arithmetic Shift Left	(exp1 x 2^count)
SHR	exp SHR count	Arithmetic Shift Right	(exp1 / 2^count)
NOT	NOT exp	Bitwise complement	
AND	exp1 AND exp2	Bitwise AND	
OR	exp1 OR exp2	Bitwise inclusive OR	
XOR	exp1 XOR exp2	Bitwise exclusive OR	

```
          ORG   H'F100
val       EQU   H'201
a         DC    val AND 255             ; a = H'F100, @a = H'01
b         DS    1                       ; b = H'F101, @b = H'01

mem2      EQU   a - 5                   ; mem2 = H'0000F0FB
c         DC    (mem2 SHR 4) OR H'FF00  ; c = H'F102, @c = H'FF0F
```

2.9.5 Relation Operators

These operators perform relation operations on the operands. They would normally be used with conditional assembly directives.

Operators	Syntax	Meaning
LT	exp1 LT exp2	Less Than
GT	exp1 GT exp2	Greater Than
LE	exp1 LE exp2	Less Than or Equal To
GE	exp1 GE exp2	Greater Than or Equal To
EQ	exp1 EQ exp2	Equal To
NE	exp1 NE exp2	Not Equal To

All operators return -1 for TRUE and 0 for FALSE.

2.9.6 Logic Operators

These operators perform logic operations in the operands.

Operators	Syntax	Meaning
LOGIC	LOGIC exp	Convert non-zero to -1
NOT	NOT exp	Logical complement
AND	exp1 AND exp2	Logical AND
OR	exp1 OR exp2	Logical inclusive OR

TRUE is equivalent to -1 and FALSE is equivalent to 0. All operators are effectively bitwise. Non-zero values need to be converted to -1 with the LOGIC operator, before use.

```
         ORG    H'F100
a        DS    1                       ; a = H'F100
b        DS    1                       ; b = H'F101
A        EQU   H'201                   ; A = H'201
B        EQU   (a SHR 4) OR H'FF10     ; B = H'FF10

mem2     EQU   ((A GT B) AND  a) OR ((A LT B) AND b)        ; mem2 = H'F101
```

2.9.7 Assignment Operators

The assembler supports the following assignment operators:

- Word (HWRD, LWRD)
- Integer (INT)
- Fractional (FRBIT)
- String (PASC)

Word Assignment

Syntax:	HWRD *exp*	Assign High Word of *exp*
	LWRD *exp*	Assign Low Word of *exp*

Usage: Fractionals and Integers are automatically rounded to 16-bits when they are assigned to a memory location. The HWRD and LWRD operators assign the high and low order words of the 32-bit internal representation, without rounding. These operators act on the bit representations of the number. It is important to take into consideration how the two different number types (fractional and integer) are represented internally, as these representations are very different. See section 2.7.3

Integer Assignment

Syntax:	INT exp	Assign the Integer part of the floating point number exp.

Usage: The INT operator views the operand as a floating point number. It then rounds it up (or down) to the nearest whole number, and assigns it as an integer.

Fractional Assignment

Syntax:	FRBIT exp	Cast the fractional bit pattern of *exp* to an Integer type.

Usage: The FRBIT operator views the operand as a fraction. It then casts the fraction to an integer, maintaining the same bit pattern. Floating point numbers outside the fractional range are rounded to the nearest whole value (as in the INT operator), with a warning of the range violation.

String Assignment

Syntax: PASC string Convert ‘\0’ to space in string.

Usage: The PASC operator is often used on unpacked strings, in combination with the %OUT directive, to generate output during the assembly process (see section 2.13.1).

Program - “assign.xap”

```
        MODULE   assignment_example
        .CONST
s1      DC       'Hello'           ; s1 = '\0H\0e\0l\0l\0o'
s2      DC       PASC('Hello')     ; s2 = ' H e l l o'
s3      DC       P"Hello"          ; s3 = 'Hello\0'

v1      EQU      H'12345678        ; v1 = H'12345678
v2      EQU      HWRD(v1)          ; v2 = H'1234
v3      EQU      LWRD(v1)          ; v3 = H'5678
v4      EQU      INT(34.23)        ; v4 = D'34
v5      EQU      INT(34.67)        ; v5 = D'35
v6      EQU      FRBIT(0.5)        ; v6 = H'40000000
        ENDMOD   assignment_example
```

Listing - “assign.lst”

```
################################################################################
#                                                                              #
# XAP Assembler - Vx.xx  (xx/xx/xx)                Tue May 19 11:06:23 1998  #
#                                                                              #
# COMMAND LINE : xapasm assign -list                                           #
#                                                                              #
#                                                                              #
#                                                                              #
# (c) Cambridge Consultants Ltd 1998                                           #
#                                                                              #
################################################################################


File: assign.xap

    1              :                     MODULE     assignment_example
    2              :                     .CONST
    3 0000 R  0048 : s1                  DC         'Hello'                        ; s1 = '\0H\0e\0l\0l\0o'
    . 0001 R  0065 :
    . 0002 R  006C :
    . 0003 R  006C :
    . 0004 R  006F :
    4 0005 R  2048 : s2                  DC         PASC('Hello')                  ; s2 = ' H e l l o'
    . 0006 R  2065 :
    . 0007 R  206C :
    . 0008 R  206C :
    . 0009 R  206F :
    5 000A R  4865 : s3                  DC         P"Hello"                       ; s3 = 'Hello\0'
    . 000B R  6C6C :
    . 000C R  6F00 :
    6              :
    7              : v1                  EQU        H'12345678                     ; v1 = H'12345678
    8              : v2                  EQU        HWRD(v1)                       ; v2 = H'1234
    9              : v3                  EQU        LWRD(v1)                       ; v3 = H'5678
   10              : v4                  EQU        INT(34.23)                     ; v4 = D'34
   11              : v5                  EQU        INT(34.67)                     ; v5 = D'35
   12              : v6                  EQU        FRBIT(0.5)                     ; v6 = H'40000000
   13              :                     ENDMOD     assignment_example


SUMMARY
-------

MODULES : 1

assignment_example              : Start Add = H'0000, Size = H'0000

LITVALS : 0

CODE SYMBOLS : 0

DATA SYMBOLS : 3

assignment_example.s1           : Address = H'0000
assignment_example.s2           : Address = H'0005
assignment_example.s3           : Address = H'000A

DATA STRUCTURE SYMBOLS : 0

EQUATE SYMBOLS : 6

assignment_example.v1           : Value = H'12345678, D'305419896
assignment_example.v2           : Value = H'00001234, D'4660
assignment_example.v3           : Value = H'00005678, D'22136
assignment_example.v4           : Value = H'00000022, D'34
assignment_example.v5           : Value = H'00000023, D'35
assignment_example.v6           : Value = H'40000000, D'1073741824

REDEFINABLE EQUATE SYMBOLS : 0

STRING EQUATE SYMBOLS : 0

SEGMENTS : 4

$??CODE                         : Segment Not Used
$??DATA                         : Segment Not Used
$??VAR                          : Segment Not Used
$??CONST                        : Segment Start Add = H'0000, Size = H'000D


ASSEMBLY
--------

NO ERRORS FOUND
WARNINGS : 1

Warning     : No Code generated
```

2.9.8 Operator Precedence

Expressions are evaluated according to the following rules:

- Operations of highest precedence are performed first.
- Operations of equal precedence are performed from left to right
- The order of evaluation can be over-ridden by using parentheses.
- Operations in parentheses are always performed before any adjacent operations.

The order of precedence is shown below, operators on the same line have equal precedence and are evaluated left to right.

Precedence	Operators
(Highest)	
10	LENGTH, (, < (text-lit), ! (char-lit), & (substitute)
9	INDEX
8	LWRD, HWRD,FREXP, FRMAN, LDEXP, FEXP, FMAN, E
7	+, - (unary)
6	*, /, MOD, SHL, SHR
5	+, - (binary), ABS
4	EQ, NE, LT, LE, GT, GE
3	LOGIC, NOT (bitwise / logical), PASC, INT, FRBIT
2	AND (bitwise / logical)
1	OR, XOR (bitwise / logical)
0	), > (text-lit)
(Lowest)	

Symbols, Constants and Strings have precedence 0 when evaluated as part of an expression.

2.9.9 Expressions

Expressions consist of several operands (constants, labels, assembler variables, or expressions) combined (using operators) to describe a value or memory location. They are often used to describe values that would be difficult or inconvenient to specify directly in the source code. Expressions are evaluated at assembly time.

2.10 Structures

2.10.1 Defining Structure Types

The STRUC and ENDS directives mark the beginning and end of a type definition for a structure. The names given to fields must be unique within the structure in which they are defined.

Structure types act as templates, allowing structures to be defined. They also allow offsets within the structure to be referred to.

```
name  STRUC
field_definitions
ENDS
```

Structure type names can be of any length and :

- Start with ‘A..Z’, ‘a..z’, ‘_’ , ‘?’ or ‘$’.
- Contain only the following characters: ‘A..Z’, ‘a..z’, ‘0..9’, ‘_’ or ‘?’
- Are case insensitive.
- Conform to the following templates :-

  $?name - ‘C’ automatically generated global structure type. (reserved)
  ?name - ‘C’ automatically generated local structure type. (reserved)
  $name - global structure type.
  name - local structure type.

- Must not match a DIRECTIVE, INSTRUCTION or MACRO name (case insensitive comparison). Structure types share the same “Name Space” as MACROs.

Field definitions have the following format :

```
name  DS    expression
```

Field names can be of any length and :

- Start with ‘A..Z’, ‘a..z’, ‘_’ or ‘?’.
- Contain only the following characters: ‘A..Z’, ‘a..z’, ‘0..9’, ‘_’ or ‘?’
- Conform to the following templates :-

  ?name - ‘C’ automatically generated local field name. (reserved)
  name - user generated field name.

- Are case sensitive.

Each structure definition has its own “Name Space”. Field declaration names must be unique within this “Name Space”.

A structure, structure type and structure field can all have the same name.

2.10.2 Defining Structures

Structures are defined as follows:

```
name  structure_type [< expression_list >]
```

The expressions contained in the optional expression list (enclosed by the < > operators) are assigned to the fields within the structure when the structure is defined. The list contains an entry for each field, with entries separated by commas.

Each field entry can be evaluated as a series of expressions by nesting it in another set of < > operators. Only one level of nesting is allowed.

Structure names have the same restrictions as label and assembler variable names (see section 2.5.3) and share the same name space.

The continuation character (‘\’) can be used to split the expression_list up over a number of lines to aid readability.

2.10.3 Using Structures

Structure fields can be accessed in the same way as normal variables using the syntax *structure_name.field_name*. For example:

fred.jim	- Field jim in structure fred
^main.fred.jim	- Field jim in structure fred in module main
$fred.jim	- Field jim in global structure fred

Operators

LENGTH	- gives the length of a structure or structure field in words.
INDEX	- gives the offset of a field in words.

The LENGTH and INDEX operators are only valid on structure types.

```
fred            STRUC
jim             DS    2
fred            DS    1
                ENDS

                .CONST
fred            fred <<10,19>, 23>      ; creates structure fred of structure type
                                        ; fred

v1              =     fred.fred         ; address of field fred in structure fred.
v2              =     fred              ; address of structure fred
v3              =     INDEX(fred.fred)  ; offset of field fred in structure type
                                        ; fred
v4              =     LENGTH(fred.fred) ; size of field fred in structure type
                                        ; fred
v5              =     LENGTH(fred)      ; size of structure type fred
```

As well as defining tables, structures provide an easy way of representing stacks and accessing variables stored on a stack.

2.10.4 Embedded Equates

In addition to field definitions, structures can also contain comments, and numeric/string equates (EQU and EQU <…>). Equates defined inside a structure have the same scope and visibility as equates defined outside a structure. However, there is one significant difference. As structures are processed during pass 1 of the assembly process, an equate defined within a structure cannot contain a forward reference, whereas equates defined outside structures can contain a single level of forward reference.

```
T_Field         STRUC

;
; crop is either C_Maize, C_Corn or C_Barley
;

C_Maize         EQU   H'0001
C_Corn          EQU   H'0002
C_Barley        EQU   H'0003

crop            DS    1

;
; size can be either C_Small or C_Large
;

C_Small         EQU   H'0000
C_Large         EQU   H'0001

size            DS    1

                ENDS
```

Program - “struc.xap”

```
                MODULE    time

time            STRUC
string          DS        20
day             DS        1
month           DS        1
year            DS        1
time            DS        2
                ENDS

date            STRUC
day             DS        1
month           DS        1
year            DS        1
                ENDS

fred_s          STRUC
var1            DS        1
var2            DS        1
                ENDS

                .CONST
time            time      <<P'Monday 18th June 1995', 0>,     \
                          18,                                 \
                          6,                                  \
                          1995,                               \
                          <2312, 54>>

yday            date      <9,4,1987>                    ; assign constant

                .CODE
                ld        al,@yday.day
                ld        x,#yday
                ld        ah,@(INDEX(date.year), x)

                ld        y,#time.time
                ld        x,#(time.time + 1)

                sub       sp,#(LENGTH fred_s)           ; allocate space on stack

                ld        al,@(INDEX(fred_s.var1), sp)  ; access var1
                add       sp,#(LENGTH fred_s)           ; return space

                ENDMOD    time
```

Listing - “struc.lst”

```
##############################################################################
#                                                                            #
# XAP Assembler - Vx.xx  (xx/xx/xx)                Tue May 19 13:53:58 1998  #
#                                                                            #
# COMMAND LINE : xapasm struc -list                                          #
#                                                                            #
#                                                                            #
#                                                                            #
# (c) Cambridge Consultants Ltd 1998                                         #
#                                                                            #
##############################################################################


File: struc.xap

    1              :                       MODULE     time
    2              :
    3 S            : time                  STRUC
    4 S            : string                DS         20
    5 S            : day                   DS         1
    6 S            : month                 DS         1
    7 S            : year                  DS         1
    8 S            : time                  DS         2
    9 S            :                       ENDS
   10              :
   11 S            : date                  STRUC
   12 S            : day                   DS         1
   13 S            : month                 DS         1
   14 S            : year                  DS         1
   15 S            :                       ENDS
   16              :
   17 S            : fred_s                STRUC
   18 S            : var1                  DS         1
   19 S            : var2                  DS         1
   20 S            :                       ENDS
   21              :
   22              :                       .CONST
   23 0000 R  4D6F : time                  time       <<P'Monday 18th June 1995', 0>,     \
    .                                                 18,                                 \
    .                                                 6,                                  \
    .                                                 1995,                               \
    .                                                 <2312, 54>>
    . 0001 R  6E64 :
    . 0002 R  6179 :
    . 0003 R  2031 :
    . 0004 R  3874 :
    . 0005 R  6820 :
    . 0006 R  4A75 :
    . 0007 R  6E65 :
    . 0008 R  2031 :
    . 0009 R  3939 :
    . 000A R  3500 :
    . 000B R  0000 :
    . 0014 R  0012 :
    . 0015 R  0006 :
    . 0016 R  07CB :
    . 0017 R  0908 :
    . 0018 R  0036 :
   28              :
   29 0019 R  0009 : yday                  date       <9,4,1987>                      ; assign constant
    . 001A R  0004 :
    . 001B R  07C3 :
   30              :
   31              :                       .CODE
   32 0000 C 01915 :                       ld         al,@yday.day
   33 0001 C 01918 :                       ld         x,#yday
   34 0002 C 00212 :                       ld         ah,@(INDEX(date.year), x)
   35              :
   36 0003 C 0171C :                       ld         y,#time.time
   37 0004 C 01818 :                       ld         x,#(time.time + 1)
   38              :
   39 0005 C 0025C :                       sub        sp,#(LENGTH fred_s)             ; allocate space on stack
   40              :
   41 0006 C 00017 :                       ld         al,@(INDEX(fred_s.var1), sp)    ; access var1
   42 0007 C 0023C :                       add        sp,#(LENGTH fred_s)             ; return space
   43              :
   44              :                       ENDMOD     time


SUMMARY
-------

MODULES : 1

time                            : Start Add = H'0000, Size = H'0008

LITVALS : 0

CODE SYMBOLS : 0

DATA SYMBOLS : 0
```

Listing - “struc.lst” (continued)

```
DATA STRUCTURE SYMBOLS : 10

time.time                       : Address = H'0000, Size = H'0019, D'25
time.time.string                : Address = H'0000, Size = H'0014, D'20
time.time.day                   : Address = H'0014, Size = H'0001, D'1
time.time.month                 : Address = H'0015, Size = H'0001, D'1
time.time.year                  : Address = H'0016, Size = H'0001, D'1
time.time.time                  : Address = H'0017, Size = H'0002, D'2
time.yday                       : Address = H'0019, Size = H'0003, D'3
time.yday.day                   : Address = H'0019, Size = H'0001, D'1
time.yday.month                 : Address = H'001A, Size = H'0001, D'1
time.yday.year                  : Address = H'001B, Size = H'0001, D'1

EQUATE SYMBOLS : 0

REDEFINABLE EQUATE SYMBOLS : 0

STRING EQUATE SYMBOLS : 0

SEGMENTS : 4

$??CODE                         : Segment Start Add = H'0000, Size = H'0008
$??DATA                         : Segment Not Used
$??VAR                          : Segment Not Used
$??CONST                        : Segment Start Add = H'0000, Size = H'001C



ASSEMBLY
--------

NO ERRORS FOUND
NO WARNINGS
```

2.11 Macros

Macros are a series of statements that are assigned to a symbol name (and optionally parameters) so that the symbol can be used in place of the statements.

Macros are processed at assembly time. They can simplify writing source code by allowing the user to substitute mnemonic names for simple inline functions and repetitive code. By changing a macro, a programmer can change the effect of statements throughout the source code.

2.11.1 Using Macros

Macros enable you to assign a symbolic name to a block of source statements, and then to use that name in your source file to represent the statements. Parameters can also be defined to represent arguments passed to the macro.

Macro expansion is a text-processing function that occurs at assembly time. Each time the assembler encounters the text associated with a macro name, it replaces that text with the text of the statements in the macro definition. Similarly, the text parameter names are replaced with the text of the corresponding actual arguments. When used the text parameter name must be prefixed with the substitute character ‘&’.

Macros are global. A macro can be defined anywhere in the source code as long as the definition precedes the first source line that calls the macro. Macros and equates are often kept in a separate file and made available to the program through an INCLUDE directive at the start of the source code.

All macros are expanded on the first pass of the assembler. Macros are expanded on every occurrence of the macro name, so they can increase the length of the executable file if called repeatedly.

2.11.2 Defining Macros

The MACRO and ENDMAC directives are used to define macros. MACRO designates the beginning of the macro block and ENDMAC designates the end.

```
name        MACRO [parameter [,parameter] ...]
            statements
            ENDMAC
```

Macro names can be of any length and :

- Start with ‘A..Z’, ‘a..z’, ‘_’ or ‘?’.
- Contain only the following characters: ‘A..Z’, ‘a..z’, ‘0..9’, ‘_’ or ‘?’
- Can be used to replace INSTRUCTIONs (case insensitive).
- Are case insensitive.
- Must not match a DIRECTIVE, or structure type name (shares the same “Name Space” with structure types).

The name must be unique and can be used later in the source code to invoke the macro.

The parameters (sometimes called dummy parameters) are names that act as placeholders for values to be passed as arguments to the macro when it is called. Up to 10 parameters can be specified and they must all fit on one line. If you give more than one parameter, you must separate them with commas, spaces, or tabs.

Any valid assembler statement may be placed within a macro, including statements that call or define other macros. Any number of statements can be used. The parameters can be used any number of times in the statement. Macros can be nested, redefined, or used recursively.

The assembler assembles the statements in a macro only if the macro is called, and only at the point in the source file from which it is called. The macro definition itself is never assembled.

A macro definition can include the LOCAL directive, which lets you define labels used only within a macro, or the EXITM directive, which allows you to exit from a macro before all the statements in the block are expanded.

2.11.3 Calling Macros

A macro call directs the assembler to copy the statements of the macro to the point of the call and to replace any parameters in the macro statements with the corresponding actual arguments.

```
name [argument [,argument] ... ]
```

The name must be the name of a macro defined earlier in the source file. The arguments can be any text. For example, symbols, constants, and registers are often given as arguments. Any number of arguments can be given, but they must all fit on one line. Multiple arguments must be separated by commas, spaces, or tabs. Commas can always be used as separators; spaces and tabs may cause ambiguity if the arguments are expressions.

If a macro call has more arguments than the macro has parameters, the extra arguments are ignored. If a call has fewer arguments than the macro has parameters, any remaining parameters are replaced with a null (empty) string.

You can use conditional statements to enable macros to check for null strings or other types of arguments. The macro can then take appropriate action to adjust to different kinds of arguments.

Program - “macro.xap”

```
            .LITVALS   H'FE00


addup       MACRO ad1, ad2, ad3, result
            ld    ah,#0
            ld    al,&ad1
            add   al,&ad2
            addc  ah,#0
            add   al,&ad3
            addc  ah,#0                ; result of addition in ah and al
            st    ah,@&result          ; store the high word
            st    al,@(&result+1)      ; store the low word
            ENDMAC


            MODULE   macro_example

            .VAR
v1          DS    2
v2          DS    2

            .CODE
            addup    #1, #4, #H'231, v1
            addup    #B'101, #124, #-H'123, v2

            ENDMOD   macro_example
```

Listing - “macro.lst”

```
##############################################################################
#                                                                            #
# XAP Assembler - Vx.xx  (xx/xx/xx)               Tue May 19 13:53:40 1998  #
#                                                                            #
# COMMAND LINE : xapasm macro -list                                          #
#                                                                            #
#                                                                            #
#                                                                            #
# (c) Cambridge Consultants Ltd 1998                                         #
#                                                                            #
##############################################################################



File: macro.xap

    1              :                      .LITVALS   H'FE00
    2              :
    3              :
    4 M            : addup                MACRO      ad1, ad2, ad3, result
    5 M            :                      ld         ah,#0
    6 M            :                      ld         al,&ad1
    7 M            :                      add        al,&ad2
    8 M            :                      addc       ah,#0
    9 M            :                      add        al,&ad3
   10 M            :                      addc       ah,#0                       ; result of addition in ah and al
   11 M            :                      st         ah,@&result                 ; store the high word
   12 M            :                      st         al,@(&result+1)             ; store the low word
   13 M            :                      ENDMAC
   14              :
   15              :
   16              :                      MODULE     macro_example
   17              :
   18              :                      .VAR
   19 0000 D  ???? : v1                   DS         2
   20 0002 D  ???? : v2                   DS         2
   21              :
   22              :                      .CODE
   23              :                      addup      #1, #4, #H'231, v1
    . 0000 C 00010 :                      * ld       ah,#0
    . 0001 C 00114 :                      * ld       al,#1
    . 0002 C 00434 :                      * add      al,#4
    . 0003 C 00040 :                      * addc     ah,#0
    . 0004 C 20035 :                      * add      al,#H'231           $??0000L
    . 0005 C 00040 :                      * addc     ah,#0                       ; result of addition in ah and al
    . 0006 C 00021 :                      * st       ah,@v1                      ; store the high word
    . 0007 C 00125 :                      * st       al,@(v1+1)                  ; store the low word
    .              :                      * ENDMAC
   24              :                      addup      #B'101, #124, #-H'123, v2
    . 0008 C 00010 :                      * ld       ah,#0
    . 0009 C 00514 :                      * ld       al,#B'101
    . 000A C 07C34 :                      * add      al,#124
    . 000B C 00040 :                      * addc     ah,#0
    . 000C C 2DD34 :                      * add      al,#-H'123
    . 000D C 00040 :                      * addc     ah,#0                       ; result of addition in ah and al
    . 000E C 00221 :                      * st       ah,@v2                      ; store the high word
    . 000F C 00325 :                      * st       al,@(v2+1)                  ; store the low word
    .              :                      * ENDMAC
   25              :
   26              :                      ENDMOD     macro_example
   27              :
    .              :                      * .SEG     LITVALS
    .              :                      * ORG      H'FE00
    . FE00 R  0231 : $??0000L             * DC       H'0231


SUMMARY
-------

MODULES : 1

macro_example                   : Start Add = H'0000, Size = H'0010

LITVALS : 1

$??0000L                        : Address = H'FE00, Value = H'0231

CODE SYMBOLS : 0

DATA SYMBOLS : 2

macro_example.v1                : Address = H'0000
macro_example.v2                : Address = H'0002

DATA STRUCTURE SYMBOLS : 0

EQUATE SYMBOLS : 0

REDEFINABLE EQUATE SYMBOLS : 0

STRING EQUATE SYMBOLS : 0
```

Listing - “macros.lst” (continued)

```
SEGMENTS : 5

$??CODE                         : Segment Start Add = H'0000, Size = H'FE01
$??DATA                         : Segment Not Used
$??VAR                          : Segment Start Add = H'0000, Size = H'0004
$??CONST                        : Segment Not Used
$??LITVALS                      : Segment Start Add = H'FE00, Size = H'0001


ASSEMBLY
--------

NO ERRORS FOUND
WARNINGS : 1

Warning     : Non-zero ROM Start Address
Description : Constant ROM Start Address = H'FE00
```

2.11.4 Using Local Macro Symbols

The LOCAL directive can be used within a macro to define symbols that are available only within the defined macro.

```
LOCAL localname [,localname] ...
```

The localname is a temporary symbol name that is to be replaced by a unique symbol name when the macro is expanded. At least one localname is required for each LOCAL directive. Once declared, localname can be used in any statement within the macro definition. The name must be prefixed with the substitute character ‘&’.

The assembler creates a new actual name for localname each time the macro is expanded. The actual name has the following form:

$??*number*

The number is a hexadecimal number in the range (hex) 0000 to (hex) FFFF. You should not give other symbols names in this format, since doing so may produce a symbol with multiple definitions. Non local labels may be used in a macro, but if the macro is used more than once, the same label will appear in both expansions, and the assembler will display an error message. To avoid this problem, use only local labels (or redefinable equates) in macros.

2.11.5 Exiting from a Macro

Normally the assembler processes all the statements in a macro definition and then continues with the next statement after the macro call. However, you can use the EXITM directive to tell the assembler to terminate macro expansion before all the statements in the macro have been assembled.

When the EXITM directive is encountered, the assembler exits the macro immediately. Any remaining statements in the macro are not processed. If EXITM is encountered in a nested macro, the assembler returns to expanding the outer block.

Program - “macro2.xap”

```
three_times    MACRO    first, second
               LOCAL    loop

               IF &second GT 20
                 nop
                 EXITM                ; exit if second GT 20
               ENDIF

               IF $count LT 3
                 ld     al,#0
                 ld     x,#&first     ; clear x, al
&loop:           add    al,#&second
                 add    x,#-1
                 bne    &loop         ; loop &first times
$count           =      $count + 1    ; increment $count
               ENDIF
               ENDMAC


               MODULE   macro_example_2

$count         =     0

start:         three_times 4, 2
               three_times 5, 4
               three_times 5, 24          ; no action, 24 GT 20
               three_times 6, 6
               three_times 7, 8           ; no action, $count EQ 3
               sleep

               ENDMOD   macro_example_2
```

Listing - “macro2.lst”

```
################################################################################
#                                                                              #
# XAP Assembler - Vx.xx  (xx/xx/xx)               Tue May 19 13:53:46 1998  #
#                                                                              #
# COMMAND LINE : xapasm macro2 -list                                           #
#                                                                              #
#                                                                              #
#                                                                              #
# (c) Cambridge Consultants Ltd 1998                                           #
#                                                                              #
################################################################################



File: macro2.xap

    1 M            : three_times          MACRO      first, second
    2 M            :                      LOCAL      loop
    3 M            :
    4 M            :                      IF         &second GT 20
    5 M            :                        nop
    6 M            :                        EXITM                                 ; exit if second GT 20
    7 M            :                      ENDIF
    8 M            :
    9 M            :                      IF         $count LT 3
   10 M            :                        ld       al,#0
   11 M            :                        ld       x,#&first                     ; clear x, al
   12 M            : &loop:                 add      al,#&second
   13 M            :                        add      x,#-1
   14 M            :                        bne      &loop                         ; loop &first times
   15 M            : $count                 =        $count + 1                    ; increment $count
   16 M            :                      ENDIF
   17 M            :                      ENDMAC
   18              :
   19              :
   20              :                      MODULE     macro_example_2
   21              :
   22              : $count               =          0
   23              :
   24              : start:               three_times 4, 2
    . 0000 C 00014 :                      * ld       al,#0
    . 0001 C 00418 :                      * ld       x,#4                          ; clear x, al
    . 0002 C 00234 : $??0000:             * add      al,#2
    . 0003 C 3FF38 :                      * add      x,#-1
    . 0004 C 3FEF0 :                      * bne      $??0000                       ; loop &first times
    .              : $count               * =        $count + 1                    ; increment $count
    .              :                      * ENDMAC
   25              :                      three_times 5, 4
    . 0005 C 00014 :                      * ld       al,#0
    . 0006 C 00518 :                      * ld       x,#5                          ; clear x, al
    . 0007 C 00434 : $??0001:             * add      al,#4
    . 0008 C 3FF38 :                      * add      x,#-1
    . 0009 C 3FEF0 :                      * bne      $??0001                       ; loop &first times
    .              : $count               * =        $count + 1                    ; increment $count
    .              :                      * ENDMAC
   26              :                      three_times 5, 24                        ; no action, 24 GT 20
    . 000A C 00000 :                      * nop
    .              :                      * EXITM                                  ; exit if second GT 20
   27              :                      three_times 6, 6
    . 000B C 00014 :                      * ld       al,#0
    . 000C C 00618 :                      * ld       x,#6                          ; clear x, al
    . 000D C 00634 : $??0003:             * add      al,#6
    . 000E C 3FF38 :                      * add      x,#-1
    . 000F C 3FEF0 :                      * bne      $??0003                       ; loop &first times
    .              : $count               * =        $count + 1                    ; increment $count
    .              :                      * ENDMAC
   28              :                      three_times 7, 8                         ; no action, $count EQ 3
    .              :                      * ENDMAC
   29 0010 C 00008 :                      sleep
   30              :
   31              :                      ENDMOD     macro_example_2


SUMMARY
-------

MODULES : 1

macro_example_2                 : Start Add = H'0000, Size = H'0011

LITVALS : 0

CODE SYMBOLS : 4

macro_example_2.start           : Address = H'0000
$??0000                         : Address = H'0002
$??0001                         : Address = H'0007
$??0003                         : Address = H'000D

DATA SYMBOLS : 0

DATA STRUCTURE SYMBOLS : 0

EQUATE SYMBOLS : 0
```

Listing - “macro2.lst” (continued)

```
REDEFINABLE EQUATE SYMBOLS : 1

$count                          : Value = H'00000003, D'3

STRING EQUATE SYMBOLS : 0

SEGMENTS : 4

$??CODE                         : Segment Start Add = H'0000, Size = H'0011
$??DATA                         : Segment Not Used
$??VAR                          : Segment Not Used
$??CONST                        : Segment Not Used


ASSEMBLY
--------

NO ERRORS FOUND
NO WARNINGS
```

2.11.6 Macro Operators

Macros use the following special set of macro operators:

Operator	Definition	Use
&	Substitute operator	Macro definition
!	Literal-character operator	Macro definition
<>	Literal-text operator	Macro call
%	Expression operator	Macro call

When used in a macro definition or a macro call, these operators carry out special control operations.

Substitute Operator

Within a macro definition, the substitute operator (&) forces the assembler to replace a parameter with its corresponding actual argument value.

&parameter

The substitute operator can be used when a parameter immediately precedes or follows other characters, or even if the parameter appears in a quoted string

```
errgen          MACRO    num, msg
                .CONST
$err&num!_msg   DC       'Error &num: &msg'
                ENDMAC

                .CODE
                errgen   5,<Unreadable disk>
```

the macro is expanded to

```
$err5_msg  DC    'Error 5: Unreadable disk'
```

For complex nested macros you can use extra ampersands to delay the replacement of a parameter. In general you need to supply as many ampersands as there are nested levels.

Literal-Character Operator

Within a macro definition, the literal-character operator (!) forces the assembler to treat a specific character literally rather than as a symbol

`!`*`character`*

The literal-character operator is used with special characters such as the semicolon or ampersand when the meaning of the special character must be suppressed. It can also be used to explicitly mark the end of a parameter name in a (&) substitution.

Literal-Text Operator

When calling a macro, the literal-text operator (<>) directs the assembler to treat a list as a single string rather that as separate arguments.

```
<text>
```

The text is considered a single literal element even if it contains commas, spaces, or tabs. The literal-text operator is most often used in macro calls to ensure that values in a parameter list are treated as single elements.

The literal-text operator can also be used to force the assembler to treat special characters, such as the semicolon or the ampersand, literally.

The assembler resolves one set of angle brackets each time the parameter is used in a macro. When using nested macros, you will need to supply as many sets of angled brackets as there are levels of nesting.

```
work  1,2,3,4,5         ; pass five parameters to the macro work
work  <1,2,3,4,5>       ; pass a single parameter to the macro work
```

Expression Operator

When calling a macro, the expression operator (%) causes the assembler to treat the argument following the operator as an expression.

```
%text
```

The assembler computes the expression’s value and replaces text with the result. The expression can either be a numeric expression or a text equate. If there are additional arguments after an argument that uses the expression operator, the additional arguments must be preceded by a comma, not a space or tab.

```
printe   MACRO exp, val
         %OUT  &exp = &val
         ENDMAC


         MODULE   expression_operator_example

sym1     EQU      100
sym2     EQU      200
msg      EQU      <'Hello World'>

         .CODE
         printe   <sym1 + sym2>, %(sym1 + sym2)
         printe   msg, %(PASC(msg))

         ENDMOD   expression_operator_example
```

will produce the following result on the display

```
sym1 + sym2 = 300
msg = H e l l o  W o r l d
```

2.11.7 Macro Recursion

Macro definitions can be recursive; that is, they can call themselves. Using recursive macros is one way of doing repeated operations. The macro does a task, and then calls itself to do the task again. The recursion is repeated until a specified condition is met.

2.11.8 Nesting Macro Definitions

One macro can define another. The assembler does not process nested definitions until the outer macro has been called. Therefore, nested macros cannot be called until the outer macro has been called at least once. Macro definitions can be nested to any depth. Nesting is limited only by the amount of memory available when the source files are assembled.

2.11.9 Nesting Macro Calls

Macro definitions can contain calls to other macros. Nested macros are expanded like any other macro call, but only when the outer macro is called.

2.11.10 Redefining Macros

Macros can be redefined. The new definition automatically replaces the old definition. If you redefine a macro from within the macro itself, make sure there are no statements or comments between the ENDMAC directive of the nested redefinition and the ENDMAC directive of the original macro.

2.12 Conditional Assembly

2.12.1 Conditional Directives

Conditional assembly directives test for a specified condition and assemble a block of statements if the condition is true. The directives are:

IF	IFDIF	IFNB	ELSE
IFB	IFE	IFNDEF	.ERR
IFDEF	IFIDN	ENDIF	

The statements following the IF directive can be any valid statements, including other conditional blocks. The ELSE directive and its statements are optional. ENDIF ends the block.

```
IF condition
   statements
[ELSE
   statements]
ENDIF
```

The statements in the conditional block are assembled only if the condition specified by the corresponding IF statement is satisfied. If the conditional block contains an ELSE directive, only the statements up to the ELSE directive are assembled. The statements that follow the ELSE directive are assembled only if the condition is not met. An ENDIF directive must mark the end of any conditional-assembly block. No more than one ELSE directive is allowed for each IF statement.

IF statements can be nested up to 255 levels. A nested ELSE directive always belongs to the nearest preceding IF statement that does not have its own ELSE.

The IF and IFE directives test the value of an expression and grant assembly based on the result.

```
IF expression
IFE expression
```

The IF directive grants assembly if the value of expression is true (non-zero). The IFE directive grants assembly if the value of expression is false (0). The expression must resolve to a constant value and must not contain forward references.

The IFDEF and IFNDEF directives test whether or not a symbol has been defined and grant assembly based on the result.

```
IFDEF name
IFNDEF name
```

The IFDEF directive grants assembly only if name is a defined label, variable or symbol. The IFNDEF directive grants assembly if name has not yet been defined. The name can be any valid name. If the name is a forward reference, it is considered undefined on pass 1.

The IFB and IFNB directives test to see if a specified argument was passed to a macro and grant assembly based on the result.

```
IFB <argument>
IFNB <argument>
```

These directives are always used inside macros, and they always test whether a real argument was passed for a specified dummy argument. The IFB directive grants assembly if argument is blank. The IFNB directive grants assembly if argument is not blank. The arguments can be any name, number or expression. Angle brackets (<>) are required.

The IFIDN and IFDIF directives compare two macro arguments and grants assembly based on the result.

```
IFIDN[I] <argument1>, <argument2>
IFDIF[I] <argument1>, <argument2>
```

These directives are always used inside macros, and they test whether real arguments passed for two specified arguments are the same. The IFIDN directive grants assembly if argument1 and argument2 are identical. The IFDIF directive grants assembly if argument1 and argument2 are different. The arguments can be names, numbers, or expressions. They must be enclosed in angle brackets and separated by a comma.

The optional I at the end of the directive name specifies that the comparison should be case insensitive.

2.12.2 Forcing Errors

The .ERR directive forces an error. The directive can be placed within a conditional-assembly block to limit the errors to certain situations.

```
.ERR <text>
```

The text is displayed on the standard output device.

2.13 Miscellaneous

2.13.1 Generating Output

The %OUT directive instructs the assembler to display text on the standard output device.

```
%OUT text
```

The text can be any line of ASCII characters. If you want to display multiple lines, you must use a separate %OUT directive for each line.

The %OUT directive can be used within a macro to print the results of an expression evaluation. The PASC (Print ASCII) operator is used to convert the null characters of an unpacked string into spaces.

```
        MODULE

        %OUT    Outputs fixed text only

        ; Defining %OUT in a macro in conjunction with %
        ; allows variable information to be output
printe  MACRO   text, val
        %OUT    &text &val
        ENDMAC

sum     EQU     2+3
        printe  <sum is>, %sum

        ; Care needs to be taken with strings as the
        ; assembler evaluates them as packed and stops at
        ; the first null
        printe  <Packed Strings are OK:>, %P'Packed String'
        printe  <Unpacked Strings disappear:>, %'Unpacked String'
        printe  <Use PASC to show an>, %PASC('Unpacked String')

        ENDMOD
```

will produce the following result on the display

```
Outputs fixed text only
sum is 5
Packed Strings are OK: Packed String
Unpacked Strings disappear:
Use PASC to show an  U n p a c k e d   S t r i n g
```

2.13.2 INCLUDE files

The INCLUDE directive inserts source code from a specified file into the source file from which the directive is given.

```
INCLUDE   filespec
```

The filespec must specify an existing file containing valid assembler statements. When the assembler encounters an INCLUDE directive, it opens the specified source file and begins processing its statements. When all statements have been read, the assembler continues with the statements immediately following the INCLUDE directive.

The filespec can be given either as a file name, or as a complete or relative file specification. For relative file paths, the current directory is that of the file doing the including.

An include file is treated like any other source file. It can be used to define macros, structures and equates or contain standard code. An include file can also include other include files, allowing nesting of source files to any level.

The FILE directive is used to record the name of the current source file. This directive is used internally by the assembler to mark the end of included files, and is not normally visible to the user.

```
FILE   filespec
```

2.13.3 Original Line Number Propagation

Line numbers can be propagated from high-level languages by including them as unique symbols within the source code. The assembler will assign a code address to the symbol allowing the symbolic debugger to equate an address with an original line number. Symbols must be of the following format:

^*module*.'*file_name*'.?*Orig_line_number*.*occurrence*:

The occurrence field allows different addresses to be mapped to the same source line number. This allows files to be included more than once within a program.

2.13.4 The Continuation Character

As an aid to readability, expressions can be split over several lines by placing a continuation character '\' at the end of each partial line. The continuation character must be preceded by a space and can not have any characters following it.

The continuation character can not be used in the middle of Symbols, Instructions or Directives, and is ignored in Strings and Comments.

```
         MODULE first

a_very_long_name_for_a_macro            \
         MACRO first
         nop
         ld    y, #&first
         ENDMAC

         .CONST
a_very_long_name_for_a_var                                              \
         DC    P'Error Message: \"number too big\"', 0,     \
               P'Error Message: \"name not valid\"', 0

         .CODE

start_address_of_a_long_line:
         ld    x, #30
         a_very_long_name_for_a_macro 3

         ENDMOD
```

2.13.5 Maximum Line Length

The maximum line length any source file can have is 500 characters. This includes any comments, and continuation lines.

2.13.6 Forcing a Page Break in a Listing File

The .PAGE directive can be used to insert a 'form feed' character into the listing file.

2.14 Error Message List

When an error occurs the assembler records the type of error and position within the source file. Assembly continues until an error is detected that makes further assembly impossible, a summary of the errors is then displayed. The error summary is appended to the end of the listing file and an error log. Error messages are displayed in a format compatible with text editors to allow jumps to erroneous lines.

The following errors can occur:

Source File Errors

"No source files specified in command line",
"No output File Name given",
"Source Stack Overflow",
"Invalid File Name",
"Unable to open file",
"Unable to set the file position",
"Unable to get the file position",
"Unexpected End Of Working File",
"Can't delete FILE - FILE open or write protected"

Macro Errors

"Invalid MACRO Name",
"Invalid MACRO parameter name",
"Unknown MACRO",
"Unexpected End of MACRO definition",
"Not part of a MACRO definition",
"Parameter/Label name already defined",
"Parameter name not found",
"Too many MACRO parameters",
"Badly formed parameter",
"Substitute parameter not found",
"Not in MACRO expansion",
"Too many MACRO LOCAL labels",
"ENDMAC/EXITM not found"

Symbol Errors

"Invalid SYMBOL Name",
"Undeclared SYMBOL",
"Invalid SYMBOL type",
"Invalid SYMBOL format",
"Invalid STRUC expression",
"Invalid STRUC assignment",
"Invalid structure format",
"Invalid local symbol name",
"Invalid field size in STRUC declaration",
"Different scopes",
"Attempting to update a symbol with a different type",
"Attempting to update a symbol with a different value",
"Not a STRUC type SYMBOL",
"Not a STRUC Definition type SYMBOL",
"Not a STRUC Field type SYMBOL",
"STRUC definition has no length allocated",
"STRUC defined in the CODE Segment",
"STRUC assignment larger than field",
"STRUC too large",
"ENDS not found",
"Too many STRUC assignment parameters",
"Defined as both a MACRO and STRUC",
"Undeclared Global SYMBOL",
"No LABEL given",
"Label defined",
"Inappropriate label definition",
"Can not assign a valid Data/Address to the symbol",
“Symbol assumed to be of word size, it is used in a forward reference”

Directive Errors

"Invalid command",
"Invalid MODULE Name",
"MODULE not defined",
"MODULE already exists",
"Non-Matching MODULE name",
"ENDMOD not found"

Segment Errors

"Code in the DATA segment",
"Unable to resolve the Code Address",
"$??CODE Address must always increment",
"Location counter should increment",
"Location counter not defined",
"LITVALS assigned to CODE segment"

Conditional Assembly Errors

"Forced Error : .ERR Instruction encountered",
"Not in Conditional Assembly",
"No associated IF statement for ELSE"

System Errors

"Can't get memory",
"Store Full",
"Stack Overflow",
"Stack Underflow",
"Assembler Internal Error",
"Too Many Errors"
"No Storage Defined"

Source Errors

"Source line too long",
“Line contains characters after Continuation character ‘\’”,
"Inappropriate expression definition",
"Expression expansion too long",
"Unknown Argument",
"Invalid Argument Format",
"No arguments in command line",
"Argument is missing quote ',"",
""-o" File name not found",
""-cmdfile" File name not found",
"More than one "-o" argument",
"Nested "-cmdfile" argument"

Expression Errors

"Invalid expression",
"Lex Buffer Full",
"Unexpected token",
"Badly formed Text Literal - No '>'",
"Converting Float (not Frac) to Integer",
"More than one parameter was given",
"Operator requires two Floats",
"Operator requires two Integers",
"Operator requires an Integer",
"Operator requires two Integers or Floats",
"Operator requires a string",
"Operator requires an Integer or Float",
"Operator requires a Float",
"Can't resolve forward reference",
"Fraction outside range: -1.0 <= x < 1.0",
"Shift value out of range: 0 <= x < 32",
“Divide by zero”,
"Invalid Escape Sequence",
"Unmatched parenthesis in expression",
"Too many nested STRING EQUATEs"

Instruction Errors

"Data Address map conflict",
"Address not valid for code/data generation",
"Invalid REGISTER",
"Invalid ADDRESS/DATA operand",
"No ADDRESS or DATA operand given",
"No REGISTER specified",
"Can't resolve data",
"Branch to DATA Symbol",
"Branch to contents of CODE Symbol",
"Not an Integer Expression",
"Address overflow - Address greater than H'FFFF",
"Integer value out of range",
"Number generated is too large",
"Number out of range - greater than H'FFFF",
"Data Out of Range",
"Address Out of Range",
"CODE Symbol",
"Too many literal value substitutions",
"LITVALS Segment not defined",
"Invalid LITVALS Address",
"Non-zero ROM Start Address",
“Code does not start at beginning of ROM”,
"No Code generated"

3 LINKER SUPPORT

The assembler has two special modes of operation which are used to allow the assembler to work with an external linker. Extensive use of these facilities is made by the ‘C’ environment and the program XAPCL.

The modes are selected by command line options ‘-c’ and ‘-s’. In these modes normal error checking and file generation is suppressed. Instead the assembler produces information in a “.lif” (linker information) file.

The ‘-c’ mode produces information on declared and undeclared global symbols.
The ‘-s’ mode produces segment type information.

xapasm file1 file2 ... **-c**

The “-c” option forces the assembler to construct a list of all global symbols declared or referenced in the source code. Symbols can be labels, variables or structures (MACROs are not included). The list can be used by the linker to extract missing functions from libraries or identify undeclared variables / functions.

The linker information file has the following format:

.DECLARED
<Declared Global Symbol List>
.UNDECLARED
<Undeclared Global Symbol List>
.END

Global Symbols - “file.lif”

```
.DECLARED
$main
$testfunc
$retstat
$in
.UNDECLARED
$setup
$post
$blenter
$_sprintf
$blexit
$strncmp
$strcpy
.END
```

xapasm file1 file2 **-s**

The "-s" option forces the assembler to construct a file containing the segment information. The list can be used by the linker to automatically define segment start addresses and sizes, making the best use of the available address space.

The linker information file has the following format:

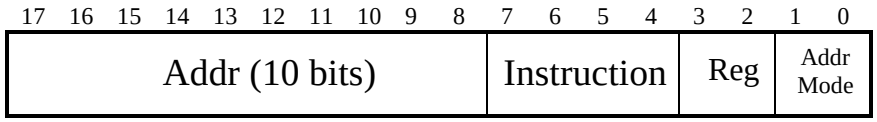
.SEGMENTS
<Segment Name> <Start Address> <Size>
.END

Segment Sizes - "file.lif"

```
.SEGMENTS
$??C_RESERVED, 0000, 10000
$??LITVALS, 0010, 00B3
$??XINITC, 0260, 0009
$??XCONST, 0F60, 1091
$??XINIT, 9000, 0009
$??XVAR, 9D00, 00CE
$??ZVAR, FE60, 0008
.END
```

4 INSTRUCTION SET

General Instruction Format:



Branch Addressing Modes

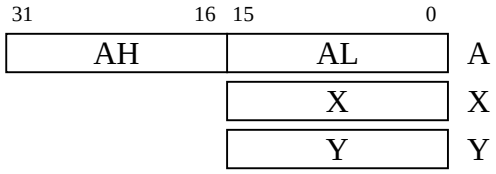
Assembler Syntax	Name	'branch addr' new PC value	Addr Mode
<nn> or label	PC relative	branch addr = PC + addr16	00
@<nn>	Direct	branch addr = @addr16	01
<nn>, X	X relative	branch addr = X + addr16	10
@(<nn>, Y)	Indexed Y	branch addr = @(Y + addr16)	11

Data Addressing Modes

Assembler Syntax	Name	'data' source/destination	Addr Mode
#<nn>	Immediate	data = addr16	00
@<nn>	Direct	data = @addr16	01
@(<nn>, X)	Indexed X	data = @(X + addr16)	10
@(<nn>, Y)	Indexed Y	data = @(Y + addr16)	11

Registers

Register	Reg
AH	00
AL	01
X	10
Y	11



Symbols

@	indicates 'contents of'.
addr16	is Addr sign extended to 16 bits.
~	NOT
&	AND
|	OR
^	XOR
<…>	replace with actual value

Instruction Set Summary

Data Transfer

LD	Load
ST	Store

Arithmetic Operations

ADD	Add
ADDC	Add with Carry
CMP	Compare
DIV	Divide
MULT	Multiply
NADD	Negated Add
SUB	Subtract
SUBC	Subtract with Carry
TST	Test

Logic Operations

AND	And
OR	Or
XOR	Exclusive Or

Shift Operations

SAL	Shift Arithmetic Left
SAR	Shift Arithmetic Right
SCL	Shift with Carry Left
SCR	Shift with Carry Right

Branch

BCC	Branch if Carry flag Clear
BCS	Branch if Carry flag Set
BEQ	Branch if Equal
BLT	Branch if Less Than
BMI	Branch if Minus
BNE	Branch if Not Equal
BPL	Branch if Plus
BRA	Branch Always
BSR	Branch to Sub-Routine

System Control

BRK	Break
NOP	No Operation
PRINT	Print
SIF	Serial Interface
SLEEP	Sleep

Operation: Register + Data → Register

Syntax: ADD <Register>, <Data Specifier>

Examples:
```
add   x, #H'3F
add   al, @(34,x)
```

Description: Add the contents of the register and the specified data and store the result in the register. Can be used for signed or unsigned, integer or fractional values.

Condition Codes:

C	S	N	Z
•	•	•	•

C	Set if a carry is generated.	C = Result[16]
S	Set if true sign of result negative.	S = Reg[15] ^ Data[15] ^ Result[16]
N	Set if result negative.	N = Result[15]
Z	Set if result zero.	Z = ~(Result[15] | ... | Result[0])

Instruction Format:

17 16 15 14 13 12 11 10 9 8	7 6 5 4	3 2	1 0
Addr (10 bits)	0 0 1 1	Reg	Addr Mode

<Data Specifier>

Assembler Syntax	Name	'data'	Addr Mode
#<nn>	Immediate	data = addr16	00
@<nn>	Direct	data = @addr16	01
@(<nn>, X)	Indexed X	data = @(X + addr16)	10
@(<nn>, Y)	Indexed Y	data = @(Y + addr16)	11

<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

Operation: Register + Data + Carry → Register

Syntax: ADDC <Register>, <Data Specifier>

Examples:
```
addc    y, #H'010A
addc    ah, @(23, y)
```

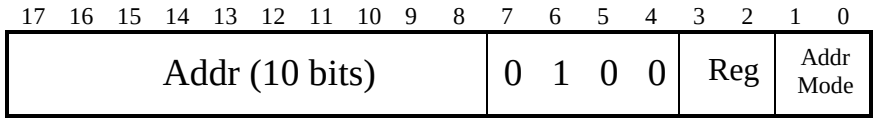
Description: Add the contents of the register, the carry flag and the specified data and store the result in the register. Can be used for signed or unsigned, integer or fractional values.

Condition Codes:

C	S	N	Z
•	•	•	•

C	Set if a carry is generated.	C = Result[16]
S	Set if true sign of result negative.	S = Reg[15] ^ Data[15] ^ Result[16]
N	Set if result negative.	N = Result[15]
Z	Set if result zero.	Z = ~(Result[15] | ... | Result[0])

Instruction Format:



<Data Specifier>

Assembler Syntax	Name	'data'	Addr Mode
#<nn>	Immediate	data = addr16	00
@<nn>	Direct	data = @addr16	01
@(<nn>, X)	Indexed X	data = @(X + addr16)	10
@(<nn>, Y)	Indexed Y	data = @(Y + addr16)	11

<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

Operation: Register & Data → Register

Syntax: AND <Register>, <Data Specifier>

Examples:
```
and   al, #H'0A
and   ah, @H'0102
```

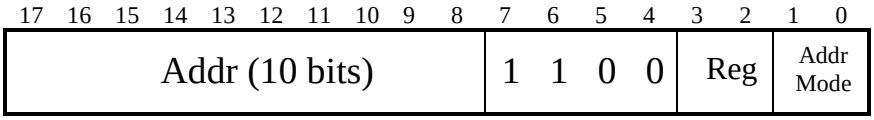
Description: AND the contents of the register with the effective address data and store the result in the register.

Condition Codes:

C	S	N	Z
-	-	•	•

C Not affected.
S Not affected.
N Set if result negative. N = Result[15]
Z Set if result zero. Z = ~(Result[15] | ... | Result[0])

Instruction Format:



<Data Specifier>

Assembler Syntax	Name	'data'	Addr Mode
#<nn>	Immediate	data = addr16	00
@<nn>	Direct	data = @addr16	01
@(<nn>, X)	Indexed X	data = @(X + addr16)	10
@(<nn>, Y)	Indexed Y	data = @(Y + addr16)	11

<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

Operation: If (C = 0) then Branch Addr → PC

Syntax: BCC <Branch Address Specifier>

Examples:
```
bcc   $ + H'0A
bcc   loop
```

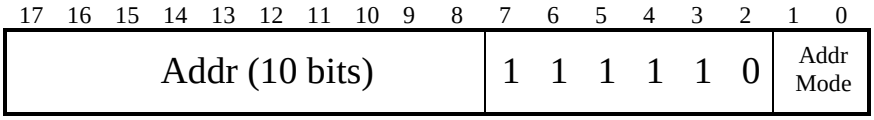
Description: If the Carry flag is Cleared, program execution continues from the branch address. If the Carry flag is Set then the next instruction is executed.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Branch Address Specifier>

Assembler Syntax	Name	'branch addr' new PC value	Addr Mode
<nn> or label	PC relative	branch addr = PC + addr16	00
@<nn>	Direct	branch addr = @addr16	01
<nn>, X	X relative	branch addr = X + addr16	10
@(<nn>, Y)	Indexed Y	branch addr = @(Y + addr16)	11

Operation: If (C = 1) then Branch Addr → PC

Syntax: BCS <Branch Address Specifier>

Examples:
```
bcs   H'0A, x
bcs   loop
```

Description: If the Carry flag is Set, program execution continues from the branch address. If the Carry flag is Cleared then the next instruction is executed.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:

17 16 15 14 13 12 11 10 9 8	7 6 5 4 3 2	1 0
Addr (10 bits)	1 1 1 1 1 1	Addr Mode

<Branch Address Specifier>

Assembler Syntax	Name	'branch addr' new PC value	Addr Mode
<nn> or label	PC relative	branch addr = PC + addr16	00
@<nn>	Direct	branch addr = @addr16	01
<nn>, X	X relative	branch addr = X + addr16	10
@(<nn>, Y)	Indexed Y	branch addr = @(Y + addr16)	11

Operation: If (Z = 1) then Branch Addr → PC

Syntax: BEQ <Branch Address Specifier>

Examples:
```
beq   @H'0A
beq   loop
```

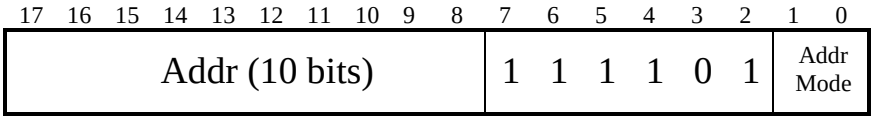
Description: If the Zero flag is Set, program execution continues from the branch address. If the Zero flag is Cleared then the next instruction is executed.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Branch Address Specifier>

Assembler Syntax	Name	'branch addr' new PC value	Addr Mode
<nn> or label	PC relative	branch addr = PC + addr16	00
@<nn>	Direct	branch addr = @addr16	01
<nn>, X	X relative	branch addr = X + addr16	10
@(<nn>, Y)	Indexed Y	branch addr = @(Y + addr16)	11

Operation: If (S = 1) then Branch Addr → PC

Syntax: BLT <Branch Address Specifier>

Examples:
```
blt   @(H'0A, y)
blt   $ - 34
```

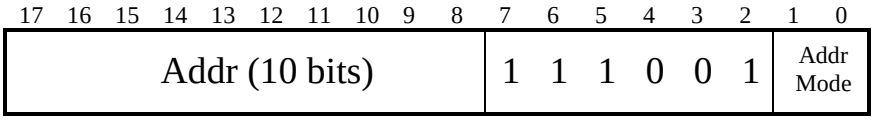
Description: If the Sign flag is Set, program execution continues from the branch address. If the Sign flag is Cleared then the next instruction is executed.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Branch Address Specifier>

Assembler Syntax	Name	'branch addr' new PC value	Addr Mode
<nn> or label	PC relative	branch addr = PC + addr16	00
@<nn>	Direct	branch addr = @addr16	01
<nn>, X	X relative	branch addr = X + addr16	10
@(<nn>, Y)	Indexed Y	branch addr = @(Y + addr16)	11

Operation: If (N = 1) then Branch Addr → PC

Syntax: BMI <Branch Address Specifier>

Examples:
```
bmi   @(H'0A, y)
bmi   $ - 34
```

Description: If the Negative flag is Set, program execution continues from the branch address. If the Negative flag is Cleared then the next instruction is executed.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:

17 16 15 14 13 12 11 10 9 8	7 6 5 4 3 2	1 0
Addr (10 bits)	1 1 1 0 1 1	Addr Mode

<Branch Address Specifier>

Assembler Syntax	Name	'branch addr' new PC value	Addr Mode
<nn> or label	PC relative	branch addr = PC + addr16	00
@<nn>	Direct	branch addr = @addr16	01
<nn>, X	X relative	branch addr = X + addr16	10
@(<nn>, Y)	Indexed Y	branch addr = @(Y + addr16)	11

Operation: If (Z = 0) then Branch Addr → PC

Syntax: BNE <Branch Address Specifier>

Examples:
```
bne   @(H'0A, y)
bne   $ - 34
```

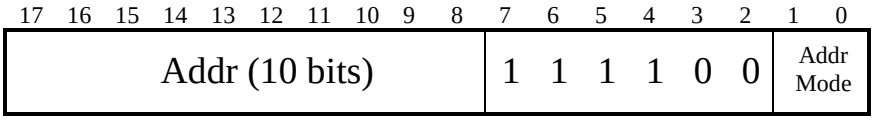
Description: If the Zero flag is Cleared, program execution continues from the branch address. If the Zero flag is Set then the next instruction is executed.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Branch Address Specifier>

Assembler Syntax	Name	'branch addr' new PC value	Addr Mode
<nn> or label	PC relative	branch addr = PC + addr16	00
@<nn>	Direct	branch addr = @addr16	01
<nn>, X	X relative	branch addr = X + addr16	10
@(<nn>, Y)	Indexed Y	branch addr = @(Y + addr16)	11

Operation: If (N = 0) then Branch Addr → PC

Syntax: BPL <Branch Address Specifier>

Examples:
```
bpl   @(H'0A, y)
bpl   start
```

Description: If the Negative flag is Cleared, program execution continues from the branch address. If the Negative flag is Set then the next instruction is executed.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:

17 16 15 14 13 12 11 10 9 8	7 6 5 4 3 2	1 0
Addr (10 bits)	1 1 1 0 1 0	Addr Mode

<Branch Address Specifier>

Assembler Syntax	Name	'branch addr' new PC value	Addr Mode
<nn> or label	PC relative	branch addr = PC + addr16	00
@<nn>	Direct	branch addr = @addr16	01
<nn>, X	X relative	branch addr = X + addr16	10
@(<nn>, Y)	Indexed Y	branch addr = @(Y + addr16)	11

Operation: Branch Addr → PC

Syntax: BRA <Branch Address Specifier>

Examples:
```
bra   H'03A, x
bra   start
```

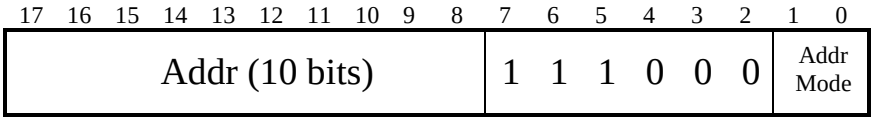
Description: Program execution continues from the branch address.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Branch Address Specifier>

Assembler Syntax	Name	'branch addr' new PC value	Addr Mode
<nn> or label	PC relative	branch addr = PC + addr16	00
@<nn>	Direct	branch addr = @addr16	01
<nn>, X	X relative	branch addr = X + addr16	10
@(<nn>, Y)	Indexed Y	branch addr = @(Y + addr16)	11

Operation: Stop for Debug

Syntax: BRK

Examples: brk

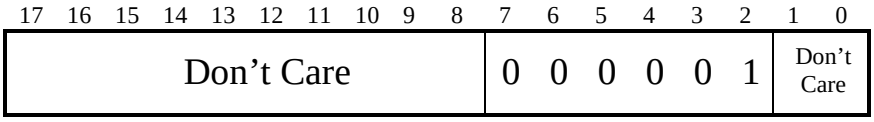
Description: Stops the processor if the RUN_STEP pin is low. The program counter (PC) is not incremented. The BRK instruction is interpreted as a NOP if the RUN_STEP pin is high.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



Operation: PC + 1 → X, Branch Addr → PC

Syntax: BSR <Branch Address Specifier>

Examples:
```
bsr   H'03A, x
bsr   start
```

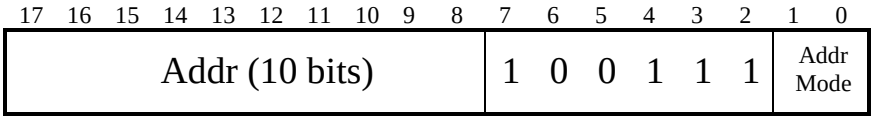
Description: The program counter (PC) + 1 is stored in the X register, and program execution then continues from the branch address.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Branch Address Specifier>

Assembler Syntax	Name	'branch addr' new PC value	Addr Mode
<nn> or label	PC relative	branch addr = PC + addr16	00
@<nn>	Direct	branch addr = @addr16	01
<nn>, X	X relative	branch addr = X + addr16	10
@(<nn>, Y)	Indexed Y	branch addr = @(Y + addr16)	11

Operation: Register - Data → Flags

Syntax: CMP <Register>, <Data Specifier>

Examples:
```
cmp   y, #H'010A
cmp   ah, @(23, y)
```

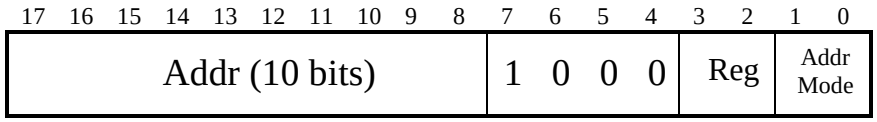
Description: Subtract the specified data from the contents of the register to set the flags. The result is not stored. Can be used for signed or unsigned, integer or fractional values.

Condition Codes:

C	S	N	Z
•	•	•	•

C	Set if a carry is generated.	C = Result[16]
S	Set if true sign of result negative.	S = Reg[15] ^ Data[15] ^ Result[16]
N	Set if result negative.	N = Result[15]
Z	Set if result zero.	Z = ~(Result[15] | ... | Result[0])

Instruction Format:



<Data Specifier>

Assembler Syntax	Name	'data'	Addr Mode
#<nn>	Immediate	data = addr16	00
@<nn>	Direct	data = @addr16	01
@(<nn>, X)	Indexed X	data = @(X + addr16)	10
@(<nn>, Y)	Indexed Y	data = @(Y + addr16)	11

<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

Operation: A >> 1 ÷ Data → AL, Remainder → AH[15:1], A[0] → AH[0]

Syntax: DIV <Data Specifier>

Example:

```
PFDIV   MACRO   data        ; signed positive fractional divide
        div     &data       ; divide A by data - result in AL
                            ;    rounding bit in AH[15]
        ENDMAC

PIDIV   MACRO   data        ; signed positive integer divide
        sal     #1          ; prepare A for divide
        div     &data       ; divide A by data - result in AL
                            ;    remainder in AH[15:1], AH[0] = 0
        ENDMAC

        ; Define some constants and variable storage areas.

        .CONST
fr_0_3  DC      HWRD(0.3), LRWD(0.3)    ; fr_0_3 (32-bit)
fr_0_5  DC      0.5                     ; fr_0_5 (16-bit)

        .VAR
fr_result DS    1           ; 16-bit fractional result
i_result  DS    1           ; 16-bit integer result

        ; Example code.

        .CODE

        ld      ah, @fr_0_3
        ld      al,@(fr_0_3 + 1)  ; A = 0.3       (32-bit)
        PFDIV   @fr_0_5           ; A = 0.6       (32-bit)
        st      al, @fr_result    ; AL = 0.6      (16-bit)

        ld      ah, #0
        ld      al, #40           ; A = 40        (32-bit)
        PIDIV   #5                ; A = 8 (32-bit)
        st      al, @i_result     ; AL = 8        (16-bit)
```

Description: The contents of the 32-bit A register (combined AH and AL) is divided by the specified data and stored in the AL register. The remainder of the division is stored in the AH register. The A register and data must contain positive signed values.

Notes: The A register must be shifted left by one, before the divide operation, for an integer divide. No shift operations are required for fractional divides.

The A register becomes a long integer or double precision fractional, prior to the divide operation. Divide will overflow if AH ≥ Data. No indication of overflow is provided.

To implement a full signed divide, the operands must be made positive before the divide and the result corrected afterwards.

The easiest way to use divide is to stick to the code given in the macros (PFDIV and PIDIV) for integer and fractional divide.

If div is used to do an integer division on its own without the shift, its operation is:

1) Divide A by 2, making a note of the fractional half bit f (the old A[0]). A can be a 32 bit unsigned value.
2) Divide result by data. MSB of data must be zero for correct operation.
3) Put result in AL, remainder in AH[15:1] and fractional bit f in AH[0]

When dividing by a constant, the shift instruction can sometimes be avoided by halving the constant.

normal code			*saves an instruction*	
sal	#1			
div	constant	→	div	(constant / 2)

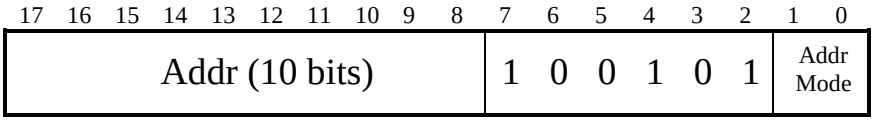
(Note: without the initial sal instruction the resultant AH[0] may be non-zero, whereas in the normal case it is always zero.)

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Data Specifier>

Assembler Syntax	Name	'data'	Addr Mode
#<nn>	Immediate	data = addr16	00
@<nn>	Direct	data = @addr16	01
@(<nn>, X)	Indexed X	data = @(X + addr16)	10
@(<nn>, Y)	Indexed Y	data = @(Y + addr16)	11

Operation: Data → Register

Syntax: LD <Register>, <Data Specifier>

Examples:
```
ld   x, #H'3F
ld   ah, @(23, y)
```

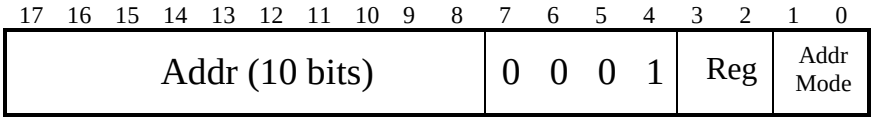
Description: Load the register with the specified data.

Condition Codes:

C	S	N	Z
-	-	•	•

C Not affected.
S Not affected.
N Set if data negative. N = Data[15]
Z Set if data zero. Z = ~(Data[15] | ... | Data[0])

Instruction Format:



<Data Specifier>

Assembler Syntax	Name	'data'	Addr Mode
#<nn>	Immediate	data = addr16	00
@<nn>	Direct	data = @addr16	01
@(<nn>, X)	Indexed X	data = @(X + addr16)	10
@(<nn>, Y)	Indexed Y	data = @(Y + addr16)	11

<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

Operation: AH + (AL x Data) → A

Syntax: MULT <Data Specifier>

Example:

```
FMULT   MACRO   data        ; signed fractional multiply
        ld      ah, #0      ; clear the AH register
        mult    &data       ; multiply AL by data
        sal     #1          ; correct for fractional multiply
        ENDMAC

IMULT   MACRO   data        ; signed integer multiply
        ld      ah, #0      ; clear the AH register
        mult    &data       ; multiply AL by data
        ENDMAC

        ; Define some constants and variable storage areas.

        .CONST
fr_0_2  DC      0.2         ; fr_0_2 = 0.2 or H'199A
fr_0_3  DC      0.3         ; fr_0_3 = 0.3 or H'2666

        .VAR
fr_result DS    1           ; 16-bit fractional result
i_result  DS    1           ; 16-bit integer result

        ; Example code.

        .CODE

        ld      al, @fr_0_2
        FMULT   @fr_0_3             ; A = 0.06      (32-bit)
        st      ah, @fr_result      ; AH = 0.06 (16-bit)

        ld      al, #23
        IMULT   #3                  ; A = 69        (32-bit)
        st      al, @i_result       ; AL = 69       (16-bit)
```

Description: The contents of the AL register is multiplied by the specified data then added to the AH register, before being stored in the A register.

Notes: The AH register should normally be cleared before a multiply operation. If non zero it adds to the result as a signed value (ie is sign extended).

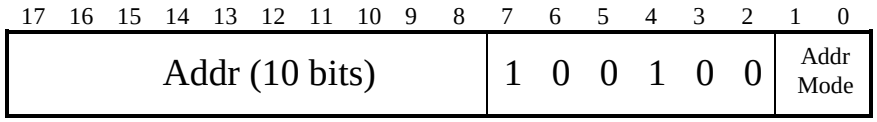
Fractional multiplies should be shifted left after the multiply operation. Integer multiplies require no correction.
Integer multiplies never overflow, even if AH is non-zero. Fractional multiply only overflows when multiplying -1.0 x -1.0. AH adds to the result as a signed fractional ÷ 2**15.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Data Specifier>

Assembler Syntax	Name	'data'	Addr Mode
#<nn>	Immediate	data = addr16	00
@<nn>	Direct	data = @addr16	01
@(<nn>, X)	Indexed X	data = @(X + addr16)	10
@(<nn>, Y)	Indexed Y	data = @(Y + addr16)	11

Operation: -Register + Data → Register

Syntax: NADD <Register>, <Data Specifier>

Examples:
```
nadd    x, #H'3F
nadd    al, @(34,x)
```

Description: Add the negated contents of the register to the specified data and store the result in the register. Can be used for signed or unsigned, integer or fractional values.

Condition Codes:

C	S	N	Z
•	•	•	•

C	Set if a carry is generated.	C = Result[16]
S	Set if true sign of result negative.	S = ~Reg[15] ^ Data[15] ^ Result[16]
N	Set if result negative.	N = Result[15]
Z	Set if result zero.	Z = ~(Result[15] | ... | Result[0])

Instruction Format:

17 16 15 14 13 12 11 10 9 8	7 6 5 4	3 2	1 0
Addr (10 bits)	0 1 1 1	Reg	Addr Mode

<Data Specifier>

Assembler Syntax	Name	'data'	Addr Mode
#<nn>	Immediate	data = addr16	00
@<nn>	Direct	data = @addr16	01
@(<nn>, X)	Indexed X	data = @(X + addr16)	10
@(<nn>, Y)	Indexed Y	data = @(Y + addr16)	11

<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

Operation: None

Syntax: NOP

Examples: nop

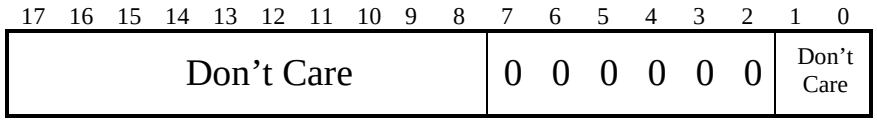
Description: Performs no operation.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



Operation: Register | Data → Register

Syntax: OR <Register>, <Data Specifier>

Examples:
```
or   al, #H'0A
or   ah, @H'0102
```

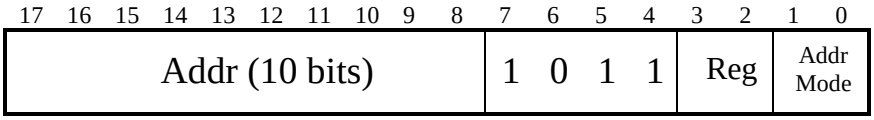
Description: OR the contents of the register with the specified data and store the result in the register.

Condition Codes:

C	S	N	Z
-	-	•	•

C Not affected.
S Not affected.
N Set if result negative. N = Result[15]
Z Set if result zero. Z = ~(Result[15] | ... | Result[0])

Instruction Format:



<Data Specifier>

Assembler Syntax	Name	'data'	Addr Mode
#<nn>	Immediate	data = addr16	00
@<nn>	Direct	data = @addr16	01
@(<nn>, X)	Indexed X	data = @(X + addr16)	10
@(<nn>, Y)	Indexed Y	data = @(Y + addr16)	11

<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

Operation: None

Syntax: PRINT <Register>, <Immediate>

Examples: print al, #H'123

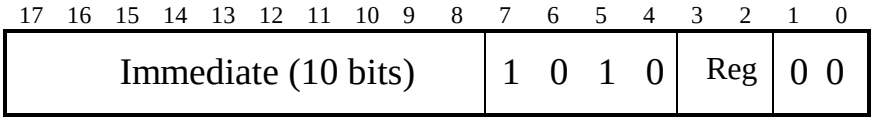
Description: A special debugging instruction only interpreted by the simulator, printing the immediate value and specified register on the display. The processor interprets the instruction as a NOP.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

Operation: Shift A left by count, (C ← [AH:AL] ← 0)

Syntax: SAL <Count Specifier>

Examples:
```
sal   #3
sal   @shift_count
```

Description: Shift the A register (combined AH and AL) left by count. Feed 0 into the least-significant bit and feed the shifted out value into the Carry flag.

Notes: A shift of 0 sets the Carry flag as if a shift of 1 had been performed, but leaves the A register un-affected.

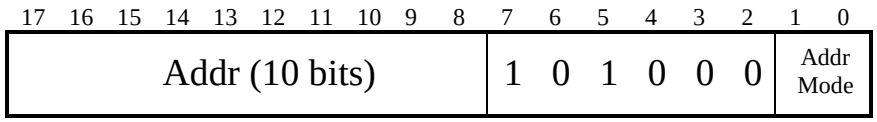
The count is masked so that the value lies within 0 .. 15.

Condition Codes:

C	S	N	Z
•	-	-	-

C Set with the last shifted out value.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Count Specifier>

Assembler Syntax	Name	'count'	Addr Mode
#<nn>	Immediate	count = addr16 & 15	00
@<nn>	Direct	count = @addr16 & 15	01
@(<nn>, X)	Indexed X	count = @(X + addr16) & 15	10
@(<nn>, Y)	Indexed Y	count = @(Y + addr16) & 15	11

Operation: Shift A right by count, (AH[15] → [AH:AL] → C)

Syntax: SAR <Count Specifier>

Examples:
```
sar   #3
sar   @shift_count
```

Description: Shift the A register (combined AH and AL) right by count. Feed AH[15] into the most-significant bit and feed the shifted out value into the Carry flag.

Notes: A shift of 0 sets the Carry flag as if a shift of 1 had been performed, but leaves the A register un-affected.

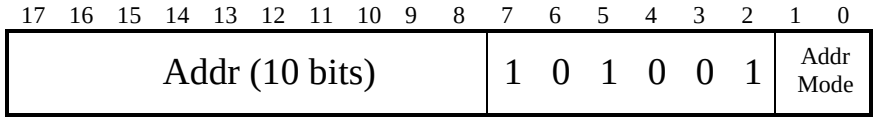
The count is masked so that the value lies within 0 .. 15.

Condition Codes:

C	S	N	Z
•	-	-	-

C Set with the last shifted out value.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Count Specifier>

Assembler Syntax	Name	'count'	Addr Mode
#<nn>	Immediate	count = addr16 & 15	00
@<nn>	Direct	count = @addr16 & 15	01
@(<nn>, X)	Indexed X	count = @(X + addr16) & 15	10
@(<nn>, Y)	Indexed Y	count = @(Y + addr16) & 15	11

Operation: Shift A left by count, (C ← [AH:AL] ← C)

Syntax: SCL <Count Specifier>

Examples:
```
scl   #3
scl   @shift_count
```

Description: Shift the A register (combined AH and AL) left by count. Feed the old Carry flag into the least-significant bit and feed the shifted out value into the Carry flag.

Notes: A shift of 0 sets the Carry flag as if a shift of 1 had been performed, but leaves the A register un-affected.

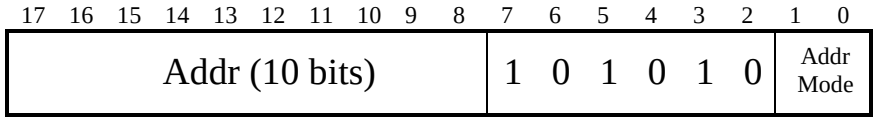
The count is masked so that the value lies within 0 .. 15.

Condition Codes:

C	S	N	Z
•	-	-	-

C Set with the last shifted out value.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Count Specifier>

Assembler Syntax	Name	'count'	Addr Mode
#<nn>	Immediate	count = addr16 & 15	00
@<nn>	Direct	count = @addr16 & 15	01
@(<nn>, X)	Indexed X	count = @(X + addr16) & 15	10
@(<nn>, Y)	Indexed Y	count = @(Y + addr16) & 15	11

Operation: Shift A right by count, (C → [AH:AL] → C)

Syntax: SCR <Count Specifier>

Examples:
```
scr   #3
scr   @shift_count
```

Description: Shift the A register (combined AH and AL) right by count. Feed the old Carry flag into the most-significant bit and feed the shifted out value into the Carry flag.

Notes: A shift of 0 sets the Carry flag as if a shift of 1 had been performed, but leaves the A register un-affected.

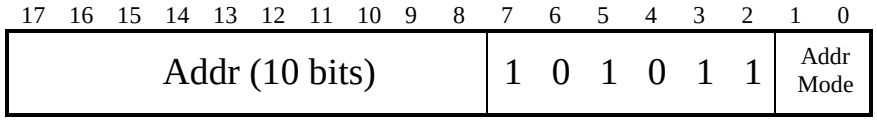
The count is masked so that the value lies within 0 .. 15.

Condition Codes:

C	S	N	Z
•	-	-	-

C Set with the last shifted out value.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Count Specifier>

Assembler Syntax	Name	'count'	Addr Mode
#<nn>	Immediate	count = addr16 & 15	00
@<nn>	Direct	count = @addr16 & 15	01
@(<nn>, X)	Indexed X	count = @(X + addr16) & 15	10
@(<nn>, Y)	Indexed Y	count = @(Y + addr16) & 15	11

Operation: Process a SIF operation

Syntax: SIF

Examples: sif

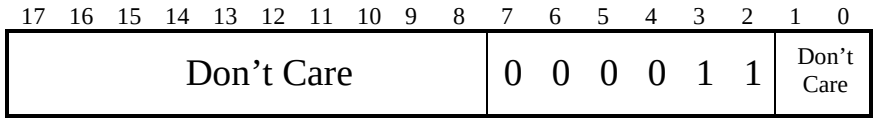
Description: Processes a SIF operation during the instruction.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



Operation: Enter Sleep mode

Syntax: SLEEP

Examples: sleep

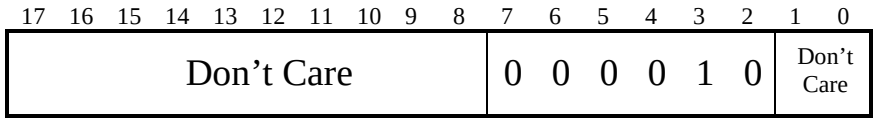
Description: Force the processor into Sleep mode.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



Operation: Register → Address

Syntax: ST <Register>, <Address Specifier>

Examples:
```
st   x, #H'3F
st   ah, @(23, y)
```

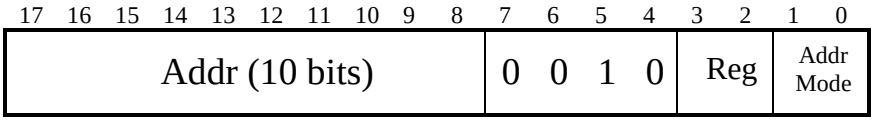
Description: Store the register with the specified address.

Condition Codes:

C	S	N	Z
-	-	•	•

C Not affected.
S Not affected.
N Set if register negative. N = Reg[15]
Z Set if register zero. Z = ~(Reg[15] | ... | Reg[0])

Instruction Format:



<Data Specifier>

Assembler Syntax	Name	'address'	Addr Mode
	Illegal mode		*00*
@<nn>	Direct	address = @addr16	01
@(<nn>, X)	Indexed X	address = @(X + addr16)	10
@(<nn>, Y)	Indexed Y	address = @(Y + addr16)	11

<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

Operation: Register - Data → Register

Syntax: SUB <Register>, <Data Specifier>

Examples:
```
sub   y, #H'010A
sub   ah, @(23,y)
```

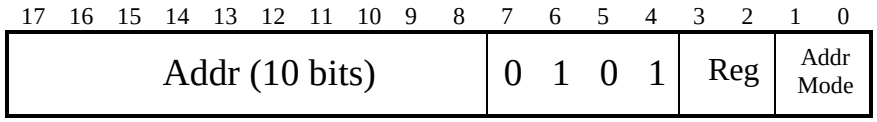
Description: Subtract the specified data from the contents of the register and store the result in the register. Can be used for signed or unsigned, integer or fractional values.

Condition Codes:

C	S	N	Z
•	•	•	•

C	Set if a carry is generated.	C = Result[16]
S	Set if true sign of result negative.	S = Reg[15] ^ Data[15] ^ Result[16]
N	Set if result negative.	N = Result[15]
Z	Set if result zero.	Z = ~(Result[15] | ... | Result[0])

Instruction Format:



<Data Specifier>

Assembler Syntax	Name	'data'	Addr Mode
#<nn>	Immediate	data = addr16	00
@<nn>	Direct	data = @addr16	01
@(<nn>, X)	Indexed X	data = @(X + addr16)	10
@(<nn>, Y)	Indexed Y	data = @(Y + addr16)	11

<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

Operation: Register - Data - Carry → Register

Syntax: SUBC <Register>, <Data Specifier>

Examples:
```
subc    y, #H'010A
subc    ah, @(23,y)
```

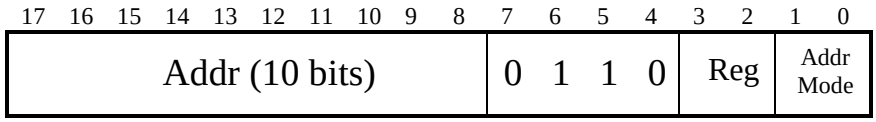
Description: Subtract the Carry and specified data from the contents of the register and store the result in the register. Can be used for signed or unsigned, integer or fractional values.

Condition Codes:

C	S	N	Z
•	•	•	•

C	Set if a carry is generated.	C = Result[16]
S	Set if true sign of result negative.	S = Reg[15] ^ Data[15] ^ Result[16]
N	Set if result negative.	N = Result[15]
Z	Set if result zero.	Z = ~(Result[15] | ... | Result[0])

Instruction Format:



<Data Specifier>

Assembler Syntax	Name	'data'	Addr Mode
#<nn>	Immediate	data = addr16	00
@<nn>	Direct	data = @addr16	01
@(<nn>, X)	Indexed X	data = @(X + addr16)	10
@(<nn>, Y)	Indexed Y	data = @(Y + addr16)	11

<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

Operation: Data → Flag

Syntax: TST <Data Specifier>

Examples: tst #13

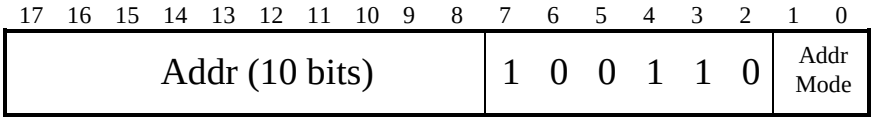
Description: Set the flags according to the specified data.

Condition Codes:

C	S	N	Z
-	-	•	•

C Not affected.
S Not affected.
N Set if data negative. N = Data[15]
Z Set if data zero. Z = ~(Data[15] | ... | Data[0])

Instruction Format:



<Data Specifier>

Assembler Syntax	Name	'data'	Addr Mode
#<nn>	Immediate	data = addr16	00
@<nn>	Direct	data = @addr16	01
@(<nn>, X)	Indexed X	data = @(X + addr16)	10
@(<nn>, Y)	Indexed Y	data = @(Y + addr16)	11

Operation: Register ^ Data → Register

Syntax: XOR <Register>, <Data Specifier>

Examples:
```
xor  al, #H'0A
xor  ah, @H'0102
```

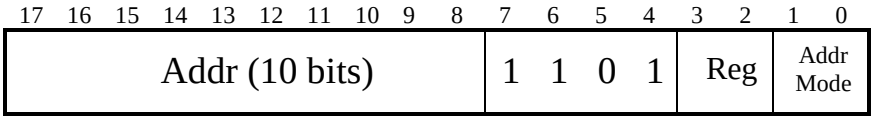
Description: XOR the contents of the register with the specified data and store the result in the register.

Condition Codes:

C	S	N	Z
-	-	•	•

C Not affected.
S Not affected.
N Set if result negative. N = Result[15]
Z Set if result zero. Z = ~(Result[15] | ... | Result[0])

Instruction Format:



<Data Specifier>

Assembler Syntax	Name	'data'	Addr Mode
#<nn>	Immediate	data = addr16	00
@<nn>	Direct	data = @addr16	01
@(<nn>, X)	Indexed X	data = @(X + addr16)	10
@(<nn>, Y)	Indexed Y	data = @(Y + addr16)	11

<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

5 KEY WORDS

Directives

%OUT	DS	FILE	IFNB
.CODE	ELSE	IF	IFNDEF
.CONST	ENDIF	IFB	INCLUDE
.DATA	ENDMAC	IFDEF	LABEL
.ERR	ENDMOD	IFDIF	LOCAL
.LITVALS	ENDS	IFDIFI	MACRO
.PAGE	EQU	IFE	MODULE
.SEG	EQU <…>	IFIDN	ORG
.VAR	EXITM	IFIDNI	STRUC
DC			

Operators

ABS	FRMAN	LENGTH	PASC
AND	GE	LOGIC	SHL
E	GT	LT	SHR
EQ	HWRD	LWRD	XOR
FEXP	INDEX	MOD	+
FMAN	INT	NE	-
FRBIT	LDEXP	NOT	*
FREXP	LE	OR	/

Instructions

ADD	BNE	NADD	SIF
ADDC	BPL	NOP	SLEEP
AND	BRA	OR	ST
BCC	BRK	PRINT	SUB
BCS	BSR	SAL	SUBC
BEQ	CMP	SAR	TST
BLT	LD	SCL	XOR
BMI	MULT	SCR	

Registers

AL	X	Y	SP
AH			

6 INDEX

P

R

S

V