



# **CYASM Assembler User's Guide Version 2.02**

May 7, 2004

• Cypress Semiconductor • Personal Communication Division •  
• 3901 North First Street • San Jose, CA 95134 • (408) 943-2600 •  
• [www.cypress.com](http://www.cypress.com) •

©Cypress Semiconductor Corp, 1998, 1999, 2000, 2001, 2002, 2003,2004.  
The information contained herein is subject to change without notice.

## Table of Contents

1.	Installing the CYASM Assembler.....	1
2.	Running CYASM .....	1
3.	The Microprocessor.....	2
	3.1 Address Spaces .....	2
	3.2 Instruction Format.....	4
	3.3 Addressing Modes.....	4
	3.4 Destination of Instruction Results.....	5
4.	Assembly Source File Syntax.....	5
5.	List File Format .....	7
6.	Assembler Directives.....	10
	BLKB - Allocate a Block of Bytes.....	11
	BLKW - Allocate a Block of Words (2 bytes).....	11
	CPU - Product specification .....	12
	CSEG - Enter CODE Segment.....	12
	DB - Define Byte.....	13
	DEFINE - Define Conditional Assembly Symbol.....	13
	DS - Define ASCII String .....	13
	DSEG - Enter DATA Segment .....	14
	DSU - Define UNICODE String.....	15
	DW - Define Word .....	15
	DWL - Define Word, Little Endian Ordering .....	15
	ELSE - Begin ELSE part of a Conditional Assembly Block.....	16
	ENDIF - End a Conditional Assembly Block.....	16
	EQU - Equate Label.....	16
	ERROR - Error Message.....	17
	FILLROM - Define unused program memory value .....	17
	IF - Begin a Conditional Assembly Block .....	17
	IFDEF - Begin a Conditional Assembly Block.....	18
	IFNDEF - Begin a Conditional Assembly Block.....	18
	INCLUDE - Include source file .....	18
	MACRO - Macro Definition Start .....	19
	ENDM - Macro Definition End .....	19
	ORG - Program Counter Origin .....	20
	XPAGEOFF - Disable XPAGE Insertion .....	20
	XPAGEON - Enable XPAGE Insertion .....	20
7.	Instruction Set.....	21

## 1. Installing the CYASM Assembler

The floppy disk included with distribution contains the cyasm.exe and sample source files. Copy the executable file to a working directory, or to a directory included in your search path.

## 2. Running CYASM

To run the assembler, from the command line enter the following command:

```
cyasm sourcefile.asm -b -t nn -pP -dD
```

The assembly language instructions reside in sourcefile, which has a ".asm" file name extension. The .asm extension does not need to be included in the command line. The full path including disk and directory names may be included in the source file name.

The following options may be included on the command line:

-b	Brief	Suppresses warning messages for operands out of range and xpage crossings.
-t nn	Tab	Sets the tab spacing in the listing file to nn, where nn is a positive integer.
-pP	Product Id	Sets the Product ID. This is the same as using the CPU directive. P is the 6-character product name from CPU Product Identification Table.
-dD	Define	Define a symbol from the command line. This is the same as using the DEFINE directive. D is the symbol string to define.

Running the assembler will result in the creation of three files: *sourcefile.rom*, *sourcefile.lst* and *sourcefile.hex*.

Generated File	Description
<i>sourcefile.rom</i>	ROM object file
<i>sourcefile.lst</i>	Listing file
<i>sourcefile.hex</i>	EPROM programming file

The following example first displays the command to assemble testfile.asm, then displays the message returned after assembling the file. A list of the output files follows.

```
>cyasm testfile -t 4

CYASM Version 2.00 Beta 01
For A and B series USB Microcontrollers
(C) 1998,1999,2000,2001,2002 Cypress Semiconductor Corp.

Complete!
>

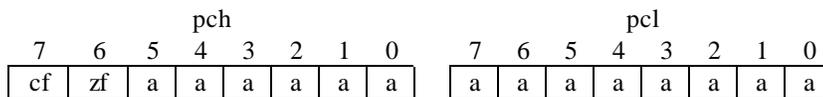
Input source:      testfile.asm
Output rom file:   testfile.rom
Output listing file: testfile.lst, with tab spacing of 4.
```

The assembler may be halted at any time during a run by pressing <Ctrl C>.

### 3. The Microprocessor

The M8 is an 8-bit microprocessor core. It supports 8-bit operations, and it has been optimized to be small and fast. There are two versions of this microprocessor — A and B version. The A version is only used in a limited number of older products, and it supports a smaller instruction set. The B version is newer and has extra instructions. The directive "CPU" is used to specify the target microprocessor core.

The Internal registers are: the accumulator "*acc*"; the index register "*X*"; the data stack pointer "*dsp*"; the program stack pointer "*psp*"; the program counter "*pc*". All registers are 8 bits wide except *pc*; *pc* is composed of two 8-bit registers (*pcl* and *pch*) that together form a 16-bit register. The lower 6 bits of *pch* and all 8 bits of *pcl* form a 14-bit address to program memory. When the *pc* is pushed on the stack, Bit 7 of the *pch* stores the carry flag ("*cf*") and bit 6 of the *pch* stores the zero flag ("*zf*").



Upon reset, *dsp* and *psp* are reset to 0x00. The *dsp* grows down, with a pre-decrement, while the *psp* grows upward with post-increment. Using a separate program stack simplifies data stack management, and provides efficient function calls.

All instructions are 1 or 2 bytes wide and are fetched from program memory, in a separate address space from data memory or IO. The second byte of an instruction is an 8-bit constant, referred to as the instruction data byte or operand. The instruction data byte is used in four different ways: as an immediate value, as a direct or offset Data RAM address, or as the lower byte of a 12-bit Program ROM address.

There are two flag bits: *zf* the zero flag and *cf* a carry / borrow flag. The flags are affected by arithmetic operations, logic and shift operations, the INDEX instruction and the JACC instruction. The manner in which each flag is changed is dependent upon the instruction being executed. Section 6, Instruction Set includes information about how each instruction affects the flags.

#### 3.1 Address Spaces

Three separate address spaces are implemented in the CYASM assembler: IO, data RAM, and program memory. The IO space is accessed through the IORD and IOWR instructions. Eight address bits are available to access the IO space. The data RAM contains the data stack, program stack, and space for variable storage. All the read and write instructions as well as instructions which operate on the stacks use data RAM. Data RAM addresses are eight bits wide, although for RAM sizes 128 bytes or smaller, not all bits are used.

The program memory is organized into 256-byte pages, such that the *pch* register contains the memory page number and the *pcl* register contains the offset into that memory page. The assembler automatically inserts an XPAGE instruction on the last location of a page to increment the page number (*pch*) in the program counter. This has the effect of moving the user assembly instruction that would have been last on one page into the first location of the next page. For two-byte instructions starting two bytes from the end of a page, a NOP is placed before the XPAGE so both bytes of the instruction are forced onto the next page. Automatic XPAGE insertion may be controlled with the XPAGEON and XPAGEOFF assembler directives.

The INDEX instruction has one operand that is the lower part of the base address of a ROM table. The lower nibble of the INDEX opcode forms the upper part of the base address, yielding a 12-bit address range. The offset into the table is taken as the value of the accumulator when the INDEX instruction is executed. The maximum readable table size when using a single INDEX instruction is limited by the range of the accumulator to 256 bytes.

An example of using an INDEX instruction is shown below.

```

tabl:      DS  "hello"      ;define a table called tabl

           MOV  A, 04
           INDEX tabl      ;fetch the 5th byte ("o") from table tabl.

```

The program memory holds the user program, as well as the data tables referenced by the INDEX instruction. INDEX, CALL (opcode 9xh), and all jump instructions have a 12-bit address range and are thereby limited to a range of 4K (see Section 3.2 Instruction Format), yet the B version supports EPROM sizes up to 8K. In order to circumvent the 4K limitation, the B version includes a second CALL instruction (opcode 5xh) that allows access to anywhere in the upper 4K of the 8K EPROM.

The XPAGE instruction is the only method other than the CALL instruction for accessing the upper 4K range of an 8K EPROM. After an XPAGE instruction has been used to cross the boundary there is then no way to return back to the lower 4K region (other than another XPAGE instruction at the top of the upper 4K range). For this reason, the CALL/RET is the suggested method for utilizing the upper 4K of code space.

During a CALL to the upper 4K, the lower 4K is not accessible by either the jump or INDEX instructions, nor is it possible to make a CALL from the upper 4K to the lower 4K. After a call into the upper 4K, access to the lower 4K is restored by the RET or RETI instructions; at that point the upper 4K is again not accessible. The following table shows allowable operation. Please note that interrupt service routines do continue to operate normally regardless of the upper/lower state at the time of the interrupt. ISRs must be located in the lower 4K range and the RETI at the end of the ISR properly returns control to either the upper or lower 4K range.

**Control flow for B version microcontroller with 8K EPROM**

Instruction Type	Low 4K to Low 4K	Low 4K to High 4K	High 4K to Low 4K	High 4K to High 4K
JACC, JC, JMP, JNC, JNZ, JZ	Yes	No	No	Yes
CALL	Yes (9x opcode)	Yes (5x opcode)	No	Yes (5x Opcode)
RET, RETI	Yes	Yes	Yes	Yes
INDEX	Yes	No	No	Yes
XPAGE	All but last page	Last page	Last Page	All but last page

The assembler examines the destination of the CALL and automatically chooses the correct opcode. If an attempt is made to do a jump, CALL or INDEX instruction that illegally crosses the 4K boundary, the assembler will flag that operation as an error.

The assembler maintains two location counters, one for data memory (RAM), and the other for program memory. The assembler switches between the two location-counters with the CSEG and DSEG directives. The assembler validates that instructions and ROM data are assembled into program memory space (CSEG) and RAM data is allocated in RAM memory space (DSEG).

## 3.2 Instruction Format

Instruction addressing is divided into two groups: (1) Logic, arithmetic and data movement functions, (2) jump and call instructions. (For the purpose of the following discussion, the INDEX opcode is grouped as a jump instruction). In the following descriptions a “0” or “1” indicates the opcode group, a “c” indicates other bits used to define opcodes, and an “a” indicates bits used to store an address or data value.

Logic, arithmetic, and data movement functions are one- or two-byte instructions. The first byte of the instruction contains the opcode, for that instruction. In two-byte instructions, the second byte contains either a data value or an address. The format for logic, arithmetic, and data movement instructions is:

Single byte instruction:

7	6	5	4	3	2	1	0
0	0	c	c	c	c	c	c

Double byte instruction:

Instruction Byte								Instruction Data Byte							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	0	c	c	c	c	c	c	a	a	a	a	a	a	a	a

All jumps instructions, plus the CALL and the INDEX instructions are 2-byte instructions. The opcode is contained in the upper 4 bits of the first instruction byte, and the destination address is stored in the remaining 12 bits. For memory sizes larger than 4 Kbytes, destination address bits above the lower 12 will be the same as those in the *pc* at the time the instruction is executed. The format for jump instruction is:

Instruction Byte								Instruction Data Byte							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
1	c	c	c	a	a	a	a	a	a	a	a	a	a	a	a

## 3.3 Addressing Modes

Three addressing modes are supported — Immediate, Direct, and Indexed. The address mode is inferred from the syntax of the assembly code. The square brackets, [] are used to denote one level of indirection. The three modes are illustrated in the following examples:

Immediate:

The immediate addressing mode is identified by a value without square brackets in the operand field. Immediate addressing causes the operand itself to be used as a value in the operation.

ADD A, 7 ;In this case the value 7 is added to the accumulator.

Direct:

The direct addressing mode is identified by a value within square brackets in the operand field. This mode causes the Data RAM value which is addressed by the operand to be used in the operation.

ADD A, [7] ;In this case the value in location 7 of the Data RAM is added to the ;accumulator.

Indexed:

The indexed addressing mode is identified by the “[X+ value]” syntax. This mode uses the value of the X-register as a base address and the operand as the offset to access locations in the Data RAM. This addressing mode is useful for indexing into a block of data within the Data RAM.

ADD A, [X+7] ;In this case, 7 is added to the current value of X-register to form the ;address. This address is then used to access the Data RAM value ;which is to be added to the accumulator.

### 3.4 Destination of Instruction Results

The result of a given instruction is stored in the entity that is placed next to the opcode in the assembly code. This allows a given result to be stored in a location other than the accumulator. Direct and Indexed addressed Data RAM locations, as well as the X-register, are additional destinations for some instructions. The AND instruction is a good illustration of this feature:

<u>Syntax</u>	<u>Operation</u>
AND A, expr	acc ← acc & k
AND A, [expr]	acc ← acc & [k]
AND A, [X + expr]	acc ← acc & [X + k]
AND [expr], A	[k] ← acc & [k]
AND [X+ expr], A	[X+ k] ← acc & [X+ k]

The ordering of the entities within the instruction determines where the result of the instruction is stored. In this example, the last two cases perform the same operation as the previous two. The difference is the destination of the instruction.

## 4. Assembly Source File Syntax

Assembly language instructions reside in files with ".asm" extensions. Each instruction has one operation on a single line, each with the format described below. The maximum line-length is 255 characters. Each keyword is separated by white spaces.

Syntax: label : MNEMONIC operands ;comment

**Label** is a case-sensitive set of alphanumeric characters and "\_" followed by a colon ":". A label may be up to 127 characters long. If used as shown in the Syntax line above, a label will be assigned a value, but labels may also be used as operands. A label is assigned the value of the current location counter (DSEG or CSEG) unless it is defined on a line with an EQU directive. Labels can be included on any line, including blank lines, but are required within an EQU directive. A label may only be defined once in an assembly program, but may be used as an operand multiple times.

If the label begins with the "." character, that label has local scope only and is recognized only between the two global labels that surround it. If the label does not begin with the "." character, that label is a global label. A label that begins with the "." character must exist between two global labels. These local labels can re-use the same names within differing global scopes.

For example:

```

Wait:   mov    a,10      ;First global label
.lp1:   dec    A
        jnz   .lp1      ;Refers to local ".lp1" above
        ret
Next:   mov    A,20     ;Second global label
.lp1:   dec    A
        jnz   .lp1     ;Refers to local ".lp1" after "Next"
        ret
Last:

```

In the above example the label ".lp1" is reused and is unique in both cases. This feature allows files to be included such that label-use conflicts are reduced.

Local labels are restricted and may be used only between global labels. For example, a programmer may not use a local label before a global label has been defined, *and* there must be at least one global label after the last local label.

**MNEMONIC** is an assembly instruction, an assembler directive, or a user defined macro name. All are defined in more detail in Section 7, Instruction Set and Section 6, Assembler Directives. There can be 0 or 1 MNEMONIC on a line of assembly code. MNEMONICS, with the exception of macro names, are case insensitive.

**Operands** either specify the addressing mode for an instruction as described in Sections 3.3 and 3.4, or are expressions that specify a value used by an instruction. The number and type of operands accepted on a line depends on the MNEMONIC on that line. Refer to Section 6, Assembler Directives and Section 7, Instruction Set for information on which operands are accepted by specific MNEMONICS. A line with no MNEMONIC must have no operands.

**Expressions** may be constructed using a number of algebraic and logical operators with any of the operand types listed in the next section. The order of precedence of the expression operators is:

- |    |                     |    |
|----|---------------------|----|
| 1. | Bitwise Complement  | ~  |
|    | Unary LSB           | >  |
|    | Unary MSB           | <  |
| 2. | Multiplication      | *  |
|    | Division            | /  |
| 3. | Addition            | +  |
|    | Subtraction         | -  |
| 4. | Bitwise Shift Left  | << |
|    | Bitwise Shift Right | >> |
| 5. | Bitwise AND         | &  |
| 6. | Bitwise XOR         | ^  |
| 7. | Bitwise OR          |    |

Parenthesis may be used to force lower precedence operations to be executed first.

**Operand Types** Labels used as operands are replaced with their defined value. Definitions may be made anywhere within the source file as described in the section on labels above. The colon that follows a label does not need to be included when used as an operand.

Constants are specified as binary, decimal, hexadecimal, or character. The radix for a number is specified by a letter following the number: b for binary, d for decimal, h for hexadecimal. If no radix is specified it is assumed to be decimal. For example 1010b, 10d, 10, and Ah are all equivalent.

Character constants are enclosed by single quotes and have the ASCII value of the character. One or two characters may be included in quotes to form an 8-bit or 16-bit value. The backslash \ is used as an escape character. To enter a single quote ' as a character, type \'. To enter a \ type \\. Some character constant examples: "A" has the value of 41h, "AB" has the value 4142h, and " \' " has the ASCII value of ' (single quote).

The dollar sign \$<sub>1</sub> is replaced by the value of the current location counter (CSEG or DSEG). For example, the instruction `JMP $` is a Jump instruction that jumps to itself.

**Comment** is anything following a semicolon ";" or a double slash "/" to the end of a line. A comment is typically used to explain the assembly code and it may be placed anywhere in the source file. Comments are ignored by the assembler; however, they are written to the listing file.

## 5. List File Format

When cyasm is run on an assembly file, a listing file with a .lst extension is created. The listing shows how the assembly program was mapped into actual memory values. It also provides a listing of errors, warnings, and a reference table of labels.

The following is a small assembly program (example.asm) and below that is its listing (example.lst).

**Sample file to assemble: *example.asm*:**

```
JMP START
DB 3FAh
                                ;only a comment on this line
ORG 20h                          ;set program counter to 20h
START: MOV a, 16d
```

**Listing of assembled file: *example.lst***

```

CYASM Version 2.00 Beta 01
(C) 1998,1999,2000,2001,2002 Cypress Semiconductor Corp.

0000 80 20 [05]          JMP START
**** Warning : '3FAh' is larger than a byte.
0002 FA  [00]          DB 3FAh
                                ;only a comment on this line
0020                                ORG 20h
                                ;set program counter to 20h
0020 19 10 [04] START: MOV a, 16d
    
```

```

Checksum = 01C3
Warnings = 1
Errors   = 0
    
```

Product: 63000, CPU Family=A, RAM=128 bytes, ROM=2048 bytes

\*\*\*\*\* RAM USAGE MAP \*\*\*\*\*

	01234567	89ABCDEF	01234567	89ABCDEF
0-1F				
20-3F				
40-5F				
60-7F				
80-9F	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
A0-BF	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
C0-DF	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
E0-FF	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX

RAM Used = 0(+) RAM Available = 128

\*\*\*\*\* ROM USAGE MAP \*\*\*\*\*

	01234567	89ABCDEF	01234567	89ABCDEF
0000-01FF	p p			
0200-03FF				
0400-05FF				
0600-07FF				
0800-09FF	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0A00-0BFF	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0C00-0DFF	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0E00-0FFF	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
1000-11FF	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
1200-13FF	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
1400-15FF	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
1600-17FF	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
1800-19FF	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
1A00-1BFF	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
1C00-1DFF	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
1E00-1FFF	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX

ROM Used = 5 ROM Available = 2043  
 F = Full Block  
 p = Partially Full Block  
 X = Unpopulated Block

\*\*\*\*\* SYMBOLIC REFERENCE TABLE \*\*\*\*\*

Value	Label	# Uses
20H	START	1

With one exception, the first column of the listing file displays either the address where the instruction is stored, or the RAM block that is allocated to it. If a plus sign "+" follows the address, the location is a RAM block address. The exception occurs when an EQU directive is assembled. In that case, the value in the first column is the value that was assigned to the label; this is further highlighted with an equal sign "=" immediately to the right of the assigned value.

The next two columns show the opcode and operand for that instruction. The exceptions to this are the define directives (see Section 6), which place defined data in each of these columns, and instructions with no operands for which column three will be left blank. The number of clock cycles required to execute the instruction is next shown in square brackets. Then the source code line corresponding to the previous information is displayed. Any warnings and errors are shown above the line that caused them.

Following the body of the listing is a checksum, a count of warnings and errors and is the symbolic reference table. Every label defined in the assembly program is included in the symbol table. The value assigned to a label is shown alongside a count of the number of times the label is used. If a label is defined by an EQU directive (see Section 6) an "=" is included between the value and label name.

As an example, look at the first line of the listing file *example.lst*. On the right is the "JMP START" from the source code. The opcode for the jump instruction (80) is placed at memory location 0000. A "20" is placed as the operand for the jump instruction at memory location 0001. The value of the operand "START" can be checked in the Symbolic Reference Table at the end of the listing. The next line of the listing is an example of a warning. In this case, the operand-value of the following define-byte (DB) assembler directive is larger than an 8-bit value. Column two of the listing line for the DB shows that the assembler used the rightmost eight bits (FA) of the operand.

## 6. Assembler Directives

The CYASM assembler allows the following assembler directives:

BLKB	Allocate a Block of Bytes
BLKW	Allocate a Block of Words (2 bytes)
CPU	Product specification
CSEG	Enter Code Segment
DB	Define Byte
DEFINE	Define Conditional Assembly Symbol
DS	Define ASCII String
DSEG	Enter Data Segment
DSU	Define UNICODE String
DW	Define Word (2 bytes)
DWL	Define Word with little endian ordering
ELSE	Begin else part of conditional assembly block
ENDIF	End a conditional assembly block
EQU	Equate label to variable value
ERROR	Error 'message'
FILLROM	Define unused program memory value
IF	Begin a conditional assembly block (based on expression result)
IFDEF	Begin a conditional assembly block (when a symbol is defined)
IFNDEF	Begin a conditional assembly block (when a symbol is not defined)
INCLUDE	Include source file
MACRO	Macro definition
ORG	Origin
XPAGEON	Xpage enable
XPAGEOFF	Xpage disable

## BLKB - Allocate a Block of Bytes

The BLKB directive allocates a block of bytes in DATA space.

Syntax:

```
label :      BLKB      expr      ;comment
```

The following is an example of allocating a block of bytes in DATA space:

**Sample file to assemble: *BLKB.asm***

```
                DSEG
                ORG 10h      ;set program counter to 10h
WORD_BUF: BLKW 5
COUNT:  BLKB 1
```

**Listing:**

```
CYASM Version 2.00 Beta 01
(C) 1998,1999,2000,2001,2002 Cypress Semiconductor Corp.
```

```
0000+          DSEG
0010+          ORG 10h      ;set program counter to 10h
0010+  WORD_BUF:  BLKW 5
001A+  COUNT:    BLKB 1
```

## BLKW - Allocate a Block of Words (2 bytes)

The BLKW directive allocates a block of words in DATA space.

Syntax:

```
label :      BLKW      expr      ;comment
```

The following is an example of allocating a block of bytes in DATA space:

**Sample file to assemble: *BLKW.asm***

```
                DSEG
                ORG 10h      ;set program counter to 10h
WORD_BUF: BLKW 5
COUNT:  BLKB 1
```

**Listing:**

```
CYASM Version 2.00 Beta 01
(C) 1998,1999,2000,2001,2002 Cypress Semiconductor Corp.
```

```
0000+          DSEG
0010+          ORG 10h      ;set program counter to 10h
0010+  WORD_BUF:  BLKW 5
001A+  COUNT:    BLKB 1
```

## CPU - Product specification

The CPU directive specifies to the assembler the resources available within the Microcontroller. The CPU directive also defines two implied symbols that can be used with the conditional assembly directives. The first symbol is the same as the Product ID listed in the CPU Product Identification Table found on page 43. The second symbol is either "ACPU" or "BCPU", based on the CPU type from the same table.

Syntax:

CPU	productName	;comment
-----	-------------	----------

The following is an example of conditional assembly code using the implied processor symbol:

```
0000          CPU 63413          ; Select the 63413
              IFDEF 63413      ; Load 14 for the 63413
0000 19 0E [04] MOV A, 14      ;
              ELSE            ; Otherwise
              ENDIF
```

## CSEG - Enter CODE Segment

The enter-code segment directive changes the current memory space to the CODE segment. Any labels that follow the CSEG directive are assigned values based on the program memory location-counter.

Syntax:

label :	CSEG	;comment
---------	------	----------

The following is an example of allocating a block of bytes in DATA space:

**Sample file to assemble:** *cseg.asm*

```
          CPU 63722
          DSEG
P_STK:   BLKB 10
APP_RAM:
          CSEG
          JMP RESET
          ORG 40h          ;set program counter to 40h
RESET:   MOV a, P_STK
          MOV PSP, A
```

### Listing:

CYASM Version 2.00 Beta 01  
(C) 1998,1999,2000,2001,2002 Cypress Semiconductor Corp.

```
0000          CPU 63722
0000+          DSEG
0000+          P_STK: BLKB 10
000A+          APP_RAM:
0000          CSEG
0000 80 40 [05] JMP RESET
0040          ORG 40h          ;set program counter to 40h
0040 19 00 [04] RESET: MOV a, P_STK
0042 60      [04]          MOV PSP, A
```

## DB - Define Byte

The define-byte directive reserves a byte of ROM and assigns the specified value to the reserved byte. This directive is useful for creating tables in ROM.

Syntax:

<b>label :</b>	<b>DB</b>	<b>operand1, operand2, ... operand(n)</b>	<b> ;comment</b>
----------------	-----------	---	------------------

The operands may be a constant or a label. The number of operands in a DB statement can be zero to as many as will fit on the source line. Below is a sample listing of an assembled set of DB directives

```
00D1 00    [00] tab1:  DB      0 , 3 , 4
00D2 03    [00]
00D3 04    [00]
00D4 06    [00]          DB      0110b
```

## DEFINE - Define Conditional Assembly Symbol

The define-conditional assembly symbol directive defines a symbol that can be tested for conditional assembly.

Syntax:

<b>DEFINE</b>	<b>symbol</b>	<b> ;comment</b>
---------------	---------------	------------------

## DS - Define ASCII String

The define-string directive stores a string of characters as ASCII values. The string must start and end with quotation marks "".

Syntax:

<b>label :</b>	<b>DS</b>	<b>'String of characters'</b>	<b> ;comment</b>
----------------	-----------	-------------------------------	------------------

The string is stored character by character in ASCII hex format. The listing file shows the first two ASCII characters on the line with the source code. The backslash character \ is used in the string as an escape character. The \ is not assembled as part of the string, but the character following it is, even if it is a \ . A quotation mark " can be entered into the middle of a string as \".

The remaining characters are shown on the following line. The string is not null-terminated. To create a null terminated string, follow the DS with a DB. Below is a sample listing for a define-ASCII string directive with a DB for a null terminated string.

```
00D8 41 42  . . .          DS      "ABCDEFGHIJK"
      43 44 45 46 47 48 49 4A 4B
00E3 00    [00]          DB      0
```

## DSEG - Enter DATA Segment

The enter-data segment directive changes the current memory space to the DATA segment. Any labels that follow the DSEG directive are assigned values based on the data memory location-counter. The enter-data segment directive accepts two optional arguments: SAVE and RESTORE.

The current location-counter normally points to Program memory, but will point to Data memory after a DSEG command is used. When the DSEG SAVE command is used, the location counter points to some Data memory location and saves that address to a temporary counter then increments by one. The DSEG location counter can be saved to a depth of 256 times.

The program continues executing, then when the DSEG RESTORE command is used, the program looks at the temporary counter and pops the previous location into the current location counter. This sets the location counter to the last location saved by the DSEG SAVE command.

Syntax:

label :	DSEG	;comment
---------	------	----------

Syntax:

label :	DSEG SAVE	;comment
---------	-----------	----------

Syntax:

label :	DSEG RESTORE	;comment
---------	--------------	----------

The following is an example of allocating a block of bytes in DATA space:

**File to assemble:** *dseg.asm*

```

        CPU 63722
        DSEG
P_STK:  BLKB 10
APP_RAM:
        CSEG
        JMP RESET
        ORG 40h           ;set program counter to 40h
RESET:  MOV a, P_STK
        MOV PSP, A

```

**Listing:**

```

CYASM Version 2.00 Beta 01
(C) 1998,1999,2000,2001,2002 Cypress Semiconductor Corp.

```

```

0000          CPU 63722
0000+          DSEG
0000+          P_STK: BLKB 10
000A+          APP_RAM:
0000          CSEG
0000 80 40 [05] JMP RESET
0040          ORG 40h           ;set program counter to 40h
0040 19 00 [04] RESET: MOV a, P_STK
0042 60      [04]          MOV PSP, A

```

## DSU - Define UNICODE String

The define-UNICODE-string directive stores a string of characters as UNICODE values with little endian byte order. The string must start and end with quotation marks "".

Syntax:

<b>label :</b>	<b>DSU</b>	<b>'String of characters'</b>	<b>;comment</b>
----------------	------------	-------------------------------	-----------------

The string is stored character by character in UNICODE format. Each character in the string is stored with the low byte followed by the high byte. The backslash character \ is used in the string as an escape character. The \ is not assembled as part of the string, but the character following it is, even if it is a \ . To enter a quotation mark " into the middle of a string, enter \".

The listing file shows the first character on the line with the source code. The remaining characters are shown on the following line. The string is not null-terminated. Below is a sample listing of an assembled define-UNICODE string directive.

```
08FE 41 00 ... DSU "ABCDE"  
42 00 43 00 44 00 45 00
```

## DW - Define Word

The define-word directive reserves two bytes of ROM and assigns the specified words to the reserved two bytes. This directive is useful for creating tables in ROM.

Syntax:

<b>label :</b>	<b>DW</b>	<b>operand1, operand2, ... operand(n)</b>	<b>;comment</b>
----------------	-----------	---	-----------------

The operands may be either a constant or a label. The number of operands in a DW statement is only limited by the length of the source line. Below is a sample listing of an assembled set of DW directives.

```
00D1 FF FE [00] tab2: DW -2  
00D3 01 DF [00] DW 01DFh  
00D5 00 11 [00] DW x  
0011= x: EQU 11h
```

## DWL - Define Word, Little Endian Ordering

The define word, little endian ordering, directive reserves two bytes of ROM and assigns the specified words to the reserved two bytes, swapping the upper and lower bytes.

Syntax:

<b>label :</b>	<b>DWL</b>	<b>operand1, operand2, ... operand(n)</b>	<b>;comment</b>
----------------	------------	---	-----------------

The operands may be either a constant or a label. The number of operands in a DW statement is only limited by the length of the source line. Below is a sample listing of an assembled set of DW directives.

```
00D1 FE FF [00] tab2: DWL -2  
00D3 DF 01 [00] DWL 01DFh  
00D5 11 00 [00] DWL x  
0011= x: EQU 11h
```

## ELSE - Begin ELSE part of a Conditional Assembly Block

The ELSE directive begins the else part of a conditional assembly block. The code between the ELSE directive and the closing ENDIF directive is assembled only if the opening IF, IFDEF, or IFNDEF directive evaluates to false. The ELSE directive does not accept parameters.

Syntax:

<b>ELSE</b>	<b>;comment</b>
-------------	-----------------

The sample below shows the ELSE part of a conditional assembly block assembled:

```
0002                                ; DEBUG symbol is not defined
                                IFDEF DEBUG      ; Load 25 for debugging
                                ELSE             ; Otherwise
0002 19 14 [04]      MOV A, 20      ; Use 20
                                ENDIF
```

## ENDIF - End a Conditional Assembly Block

The ENDIF directive ends a conditional assembly block. The ENDIF directive does not accept parameters.

Syntax:

<b>ENDIF</b>	<b>;comment</b>
--------------	-----------------

The sample below shows the ENDIF directive used in as part of a conditional assembly block assembled:

```
0002                                ; DEBUG symbol is not defined
                                IFDEF DEBUG      ; Load 25 for debugging
                                ELSE             ; Otherwise
0002 19 14 [04]      MOV A, 20      ; Use 20
                                ENDIF
```

## EQU - Equate Label

The EQUATE (EQU) directive is used to assign an integer value to a label.

Syntax:

<b>label :</b>	<b>EQU</b>	<b>operand</b>	<b>;comment</b>
----------------	------------	----------------	-----------------

The label and operand are required for an EQU directive. The operand must be a valid expression. If the expression uses the \$ (current location counter), the current memory segment location counter is used. Each EQU directive may have only one operand and if a label is defined more than once, an assembly error will occur. Below is a sample listing of an assembled set of EQU directives.

```
0000 10    [00]      DB      zz
0001 00 11 [00]      DW      yy      ;Example of how label is used
0010=      xx:      EQU     10h
0011=      yy:      EQU     11h
0010=      zz:      EQU     xx
```

## ERROR - Error Message

The ERROR Message directive causes the assembler to generate an assembly error that emits the message. This can be used in conjunction with conditional assembly to assure proper assembly.

For example, when a series of commands are programmed, the assembler may find the sequence of commands unacceptable and will therefore output the message declaring an error.

Syntax:

<b>label :</b>	<b>ERROR</b>	<b>' message '</b>	<b>;comment</b>
----------------	--------------	--------------------	-----------------

## FILLROM - Define unused program memory value

The FILLROM directive is used to force all unused bytes of program memory to a specified value.

Syntax:

<b>label:</b>	<b>FILLROM</b>	<b>value</b>	<b>;comment</b>
---------------	----------------	--------------	-----------------

Every byte of program memory, which is not otherwise used, will be assigned the value following the FILLROM directive. Only one FILLROM statement will be used to fill all unused locations.

## IF - Begin a Conditional Assembly Block

The IF directive starts a conditional assembly block. If the expression evaluates to a non-zero value, the block of code between the IF and the closing ENDIF or ELSE directive is assembled. If the expression evaluates to zero, the following code block is not assembled (nor are any assembler directives in the block assembled) and the ELSE part of the conditional block, if one exists, is assembled.

Syntax:

<b>IF</b>	<b>expression</b>	<b>;comment</b>
-----------	-------------------	-----------------

This sample code show the use of the IF directive, along with local labels enclosed by a global labels. The example shows one way to determine if the length of a fixed string is a multiple of eight.

```

0000          BLOCK:
0000 43 68 ... .S1: DS "Character String"
          61 72 61 63 74 65 72 20 53 74 72 69 6E 67
0010=          .SL1:EQU ($ - .S1)
0010 43 68 ... .S2: DS "Character String 2"
          61 72 61 63 74 65 72 20 53 74 72 69 6E 67 20 32
0012=          .SL2:EQU ($ - .S2)

          IF (.SL1 & 7H)      ; String 1 length multiple of 8?
          ELSE                ;
0022 19 02 [04]    MOV A, 2      ; Yes, use 2
          ENDF

          IF (.SL2 & 7H)      ; String 2 length multiple of 8?
          ELSE                ;
0024 19 01 [04]    MOV A, 1      ; No, load 1
          ENDF
0026          ENDBLOCK:

```

## IFDEF - Begin a Conditional Assembly Block

The IFDEF directive starts a conditional assembly block. If the symbol has been defined previously with a DEFINE directive, the block of code between the IFDEF and the closing ENDIF or ELSE directive is assembled. If the symbol has not been defined previously, the following block of code is not assembled (nor are any assembler directives in the block assembled) and the ELSE part of the conditional block, if one exists, is assembled.

Syntax:

IFDEF	symbol	;comment
-------	--------	----------

Below is a sample listing of conditional assembly using the IFDEF directive:

```

                                define DEBUG    ; Define the debug symbol
                                ifndef DEBUG     ; Load A with 1 if we are debugging
0000 19 01 [04]  MOV A, 01      ;
                                endif           ;

```

Here is the same code fragment where the DEBUG symbol is not defined:

```

;;define  DEBUG                ; Define the debug symbol
        ifndef DEBUG          ; Load A with 1 if we are debugging
        endif                 ;

```

## IFNDEF - Begin a Conditional Assembly Block

The IFNDEF directive starts a conditional assembly block. If the symbol has not been defined previously with a DEFINE directive, the block of code between the IFNDEF and the closing ENDIF or ELSE directive is assembled. If the symbol has been defined previously, the following code block is not assembled (nor are any assembler directives in the block assembled) and the else part of the conditional block, if one exists, is assembled.

Syntax:

IFNDEF	symbol	;comment
--------	--------	----------

## INCLUDE - Include source file

The INCLUDE directive is used to include additional source files into the main file being assembled.

Syntax:

label :	INCLUDE	'source_file'	;comment
---------	---------	---------------	----------

Once an INCLUDE directive is encountered the assembler reads in the new source file (source\_file) until either another INCLUDE is encountered or the end of file is found. When an end of file is encountered, the assembler resumes reading the previous file immediately following the INCLUDE directive. In other words, INCLUDE directives cause nesting of source code being assembled. The source\_file specified should contain a full path name if it does not reside in the current directory.

## MACRO - Macro Definition Start

## ENDM - Macro Definition End

The MACRO and ENDM directives are used to specify the start and end of a macro definition.

Definition Syntax:

<b>label:</b>	<b>MACRO</b>	<b>macroname</b>	<b>parm1,parm2,...,parm(n)</b>	<b>;comment</b>
	<b>macro body consisting of lines of CYASM code</b>			
	<b>ENDM</b>			

Call Syntax:

<b>label:</b>	<b>macroname</b>	<b>value1,value2,...,value(n)</b>	<b>;comment</b>
---------------	------------------	-----------------------------------	-----------------

The lines of code defined between a MACRO statement and an ENDM statement are not directly assembled into the program. Instead, they form a macro that may later be substituted into the code by a macro call. The MACRO directive is followed by the name used to call the macro as well as a list of parameters. Each of the parameters is a string that can be used in the macro body as an operand, either alone or as part of an expression. In a macro-call, each time a parameter is used in the macro body that parameter will be replaced by the corresponding value from the macro call. Any labels defined in a macro will have a #*n*, where *n* is a unique number for each macro call, appended. This makes the label unique each time the macro is used.

One example of a macro is the variable delay loop shown below.

Macro definition from source file:

```
0000          MACRO wait    delay
0000          MOV a,delay
0000          loop:  DEC a
0000          JNZ   loop
0000          ENDM
```

Macro call from source file:

```
wait 50
```

Macro instantiation from the listing file:

```
**** MACRO ****          wait    50
0100 19 32 [04]          MOV a,50
0102 25   [04] loop#1:  DEC a
0103 B1 02 [05]          JNZ   loop#1:
***END MACRO***
```

A macro must be defined earlier in the assembly file than it is called. Macro definitions may not be nested, but macros that are already defined may be used in following macro definitions.

## ORG - Program Counter Origin

The origin (ORG) directive allows the programmer to set the value of the current location during assembly. This is most often used to set the start of a table in conjunction with the define directives BLKB, BLKW, DB, DS and DW. The ORG directive only modifies the location counter of the active segment either CSEG or DSEG.

Syntax:

```
label :      ORG      operand      ;comment
```

The operand is required for an ORG directive and may be any valid constant expression, including expressions that use integer constants, labels, or “\$” (current location counter). The assembler does not keep track of previously defined areas and will not flag overlapping areas in a single source file. Below is a sample listing of an assembled set of DB directives.

```
00D1          ORG      00D1h
00D1 03      [00]      DB      3
00FD          ORG      00FDh
```

## XPAGEOFF - Disable XPAGE Insertion

The XPAGEOFF directive disables the automatic insertion of XPAGE instructions at page breaks. Most often this is useful when defining ROM tables or jump tables.

Syntax:

```
label:      XPAGEOFF
```

After the XPAGEOFF directive is encountered, the assembler will not insert XPAGE and NOP instructions at program memory page-crossings until an XPAGEON directive is encountered. The assembler defaults to XPAGE insertion on at the top of the file.

## XPAGEON - Enable XPAGE Insertion

The XPAGEON directive enables the automatic insertion of XPAGE instructions at page breaks. Often this is useful when defining ROM or JUMP tables.

Syntax:

```
label:      XPAGEON
```

The XPAGEON directive enables automatic insertion XPAGE and NOP instructions at page breaks after an XPAGEOFF directive has disabled it. The assembler defaults to XPAGE insertion on at the top of the file.

## 7. Instruction Set

The following notation will be used throughout this section of the document:

acc     Accumulator  
 expr    expression  
 k        operand value  
 X        X register

The conditional jump instructions (JC, JNC, JZ, JNZ) list two numbers in the Cycles column. The first number is the number of cycles for the instruction execution when the branch is taken. The second number, shown in parenthesis, is the number of cycles when the branch is not taken.

### ADD

### Add Without Carry

**Syntax:**

**ADD A, expr**

**ADD A, [expr]**

**ADD A, [X + expr]**

**Operation:**

$acc \leftarrow acc + k$

$acc \leftarrow acc + [k]$

$acc \leftarrow acc + [X + k]$

**Description:**     Adds a value; k, [k] or [X+ k] to the contents of the accumulator and places the result in the accumulator.

**Condition Flags:**

CF:     Set if, treating the numbers as unsigned, the result > 255; cleared otherwise.

ZF:     Set if the result is zero; cleared otherwise.

Source	Machine Code		Cycles
Format	Opcode	Operand	
ADD A, expr	01h	Immediate byte	4
ADD A, [expr]	02h	Direct address byte	6
ADD A, [X+expr]	03h	Offset byte	7

## ADC

## Add With Carry

**Syntax:****ADC A, expr****ADC A, [expr]****ADC A, [X + expr]****Operation:** $acc \leftarrow acc + k + cf$  $acc \leftarrow acc + [k] + cf$  $acc \leftarrow acc + [X + k] + cf$ 

**Description:** Adds the content of the carry bit along with the contents of the accumulator to a value; k, [k] or [X+ k] and places the result in the accumulator.

**Condition Flags:**

CF: Set if, treating the numbers as unsigned, the result &gt; 255; cleared otherwise.

ZF: Set if the result is zero; cleared otherwise.

Source	Machine Code		Cycles
Format	Opcode	Operand	
ADC A, expr	04h	Immediate byte	4
ADC A, [expr]	05h	Direct address byte	6
ADC A, [X+expr]	06h	Offset byte	7

## AND

## Bitwise AND

**Syntax:****AND A, expr****AND A, [expr]****AND A, [X + expr]****AND [expr], A****AND [X+ expr], A****Operation:** $acc \leftarrow acc \& k$  $acc \leftarrow acc \& [k]$  $acc \leftarrow acc \& [X + k]$  $[k] \leftarrow acc \& [k]$  $[X + k] \leftarrow acc \& [X + k]$ 

**Description:** A bitwise AND of a value; k, [k] or [X+ k] and the contents of the accumulator. The result is placed in either the accumulator, [k] or [X+ k] according to the field just to the right of the opcode.

**Condition Flags:**

CF: Always cleared.

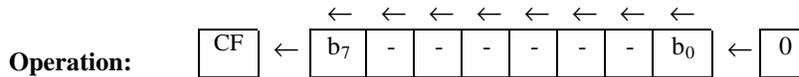
ZF: Set if the result is zero; cleared otherwise.

Source	Machine Code		Cycles
Format	Opcode	Operand	
AND A, expr	10h	Immediate byte	4
AND A, [expr]	11h	Direct address byte	6
AND A, [X+expr]	12h	Offset byte	7
AND [expr], A	35h	Direct address byte	7
AND [X+ expr], A	36h	Offset byte	8

## ASL

## Arithmetic Shift Left

**Syntax:** **ASL A** or **ASL**



**Description:** Shifts all bits of the accumulator one place to the left. The most significant bit of the accumulator is loaded into the CF flag. Bit 0 is loaded with a zero.

**Condition Flags:**

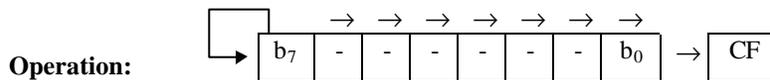
- CF: Set if the MSB of the accumulator was set, before the shift, cleared otherwise.
- ZF: Set if the result is zero; cleared otherwise.

Source Format	Machine Code		Cycles
	Opcode	Operand	
ASL	3Bh		4

## ASR

## Arithmetic Shift Right

**Syntax:** **ASR** or **ASR A**



**Description:** Shifts all bits of the accumulator one place to the right. Bit 0 of the accumulator is loaded into the CF flag. Bit 7 remains the same.

**Condition Flags:**

- CF: Set if LSB of the accumulator was set, before the shift, cleared otherwise.
- ZF: Set if the result is zero; cleared otherwise.

Source Format	Machine Code		Cycles
	Opcode	Operand	
ASR	3Ch		4

## CALL

## Call Function

**Syntax:** **CALL** address

**Operation:** [psp] ← pc  
psp ← psp + 2  
pc ← k

**Description:** Executes a jump to a subroutine starting at the address given as an operand. The Program counter (pc) is pushed onto the program stack without disturbing the data stack. The zero flag (zf) and carry flag (cf) are pushed along with the pc. The program stack pointer is incremented. The program counter is loaded with the address value.

**Condition Flags:**

CF: Carry flag unaffected.  
ZF: Zero flag unaffected.

Source	Machine Code		Cycles
Format	Opcode	Operand	
CALL addr	9xh	address byte (first 4K ROM)	10
CALL addr	5xh	address byte (second 4K ROM, B CPUs only)	10

## CMP

## Non-destructive Compare

**Syntax:**

**CMP A, expr**

**CMP A, [expr]**

**CMP A, [X + expr]**

**Operation:**

cf ← acc - k

cf ← acc - [k]

cf ← acc - [X + k]

**Description:** Subtracts a value; k, [k] or [X + k] from the contents of the accumulator and sets the flag bits. The contents of the accumulator are unaffected.

**Condition Flags:**

CF: Set if the accumulator contents < operand value; cleared otherwise.  
ZF: Set if the result is zero; cleared otherwise.

Source	Machine Code		Cycles
Format	Opcode	Operand	
CMP A, expr	16h	Immediate byte	5
CMP A, [expr]	17h	Direct address byte	7
CMP A, [X+expr]	18h	Offset byte	8

## CPL

## Complement Accumulator

**Syntax:** CPL A or CPL

**Operation:**  $acc \leftarrow \overline{acc}$

**Description:** Replace each bit in the accumulator with its complement.

**Condition Flags:**

CF: Always set.

ZF: Set if the result is zero; cleared otherwise.

Source	Machine Code		Cycles
Format	Opcode	Operand	
CPL A	3Ah		4

## DEC

## Decrement

**Syntax:**

DEC A

DEC X

DEC [expr]

DEC [X+ expr]

**Operation:**

$acc \leftarrow acc - 1$

$X \leftarrow X - 1$

$[k] \leftarrow [k] - 1$

$[X+ k] \leftarrow [X+ k] - 1$

**Description:** Subtract one from the contents of a register or Data RAM location. The field to the right of the opcode determines which entity is effected: accumulator; X register; direct or index addressed Data RAM location.

**Condition Flags:**

CF: Set if the result is -1; cleared otherwise.

ZF: Set if the result is zero; cleared otherwise.

Source	Machine Code		Cycles
Format	Opcode	Operand	
DEC A	25h		4
DEC X	26h		4
DEC [expr]	27h	direct address byte	7
DEC [X+ expr]	28h	offset address byte	8

## DI

## Disable Interrupts

**Syntax:**        **DI**                           **B CPU only**

**Operation:**    None

**Description:**   Disables interrupts.

**Condition Flags:**

CF:    Carry flag unaffected.  
ZF:    Zero flag unaffected

Source	Machine Code		Cycles
Format	Opcode	Operand	
DI	70h		4

## EI

## Enable Interrupts

**Syntax:**        **EI**                           **B CPU only**

**Operation:**    None

**Description:**   Enables interrupts.

**Condition Flags:**

CF:    Carry flag unaffected.  
ZF:    Zero flag unaffected

Source	Machine Code		Cycles
Format	Opcode	Operand	
EI	72h		4

## HALT

## Halt Execution

**Syntax:**        **HALT**

**Operation:**    None

**Description:**   Halts execution of the processor core until the occurrence of a reset - Watchdog, POR or USB

**Condition Flags:**

CF:       Carry flag unaffected.

ZF:       Zero flag unaffected

Source	Machine Code		Cycles
Format	Opcode	Operand	
HALT	00h		7

## INC

## Increment

**Syntax:**

**INC A**

**INC X**

**INC [expr]**

**INC [X+ expr]**

**Operation:**

$acc \leftarrow acc + 1$

$X \leftarrow X + 1$

$[k] \leftarrow [k] + 1$

$[X+ k] \leftarrow [X+ k] + 1$

**Description:**   Add one to the contents of a register or Data RAM location. The field to the right of the opcode determines which entity is effected: accumulator; X register; direct or index addressed Data RAM location.

**Condition Flags:**

CF:       Set if value after the increment is 0; cleared otherwise.

ZF:       Set if the result is zero; cleared otherwise.

Source	Machine Code		Cycles
Format	Opcode	Operand	
INC A	21h		4
INC X	22h		4
INC [expr]	23h	direct address byte	7
INC [X+ expr]	24h	offset address byte	8

**INDEX****Table Read****Syntax:** INDEX address**Operation:**  $\text{acc} \leftarrow \text{ROM}[\text{addr} + \text{acc}]$ **Description:** Places the contents of ROM location indexed by the sum of the accumulator and the address operand, into the accumulator. The INDEX instruction modifies one byte of RAM that is indexed by the current value of the PSP.**Condition Flags:**

- CF: Set if computed address is on a different page from the base address; cleared otherwise.  
 ZF: Set if the low byte of the computed address is 00; cleared otherwise.

Source	Machine Code		Cycles
Format	Opcode	Operand	
INDEX address	Fxh	address byte	14

**IPRET****IO Write , Pop, and Return** (A CPU only)**Syntax:** IPRET address**Operation:**  $\text{IO}[\text{address}] \leftarrow \text{acc}$  , POP acc, RET**Description:** Places the contents of the accumulator into IO location indexed the by address, then pop the accumulator from the data stack, then return from interrupt.**Condition Flags:**

- CF: Carry restored to the value that was pushed onto the program stack.  
 ZF: Zero restored to the value that was pushed onto the program stack.

Source	Machine Code		Cycles
Format	Opcode	Operand	
IPRET address	1Eh	address byte	13

## IORD

## Read IO

**Syntax:** IORD address

**Operation:**  $acc \leftarrow IO[k]$

**Description:** Places the contents of IO location indexed the by address operand into the accumulator.

**Condition Flags:**

CF: Carry flag unaffected.

ZF: Zero flag unaffected.

Source	Machine Code		Cycles
Format	Opcode	Operand	
IORD address	29h	address byte	5

## IOWR

## Write IO

**Syntax:** IOWR address

**Operation:**  $IO[k] \leftarrow acc$

**Description:** Place the contents of the accumulator into the IO location indexed by the address operand.

**Condition Flags:**

CF: Carry unaffected.

ZF: Zero unaffected.

Source	Machine Code		Cycles
Format	Opcode	Operand	
IOWR address	2Ah	address byte	5

## IOWX

## Indexed IO Write

**Syntax:** IOWX [x + address]

**Operation:** IO[x + k] ← acc

**Description:** Place the contents of the accumulator into the IO location given by the sum of the index register and the address operand.

**Condition Flags:**

CF: Carry unaffected.

ZF: Zero unaffected.

Source	Machine Code		Cycles
Format	Opcode	Operand	
IOWX address	39h	address byte	6

## JACC

## Jump Accumulator

**Syntax:** JACC address

**Operation:** pc ← acc + k

**Description:** Jump unconditionally to the address computed by the sum of the accumulator and the 12-bit address operand. The accumulator is not affected by this instruction.

**Condition Flags:**

CF: Set if computed address is on a different page from the base address; cleared otherwise.

ZF: Set if the low byte of the computed address is 00; cleared otherwise.

Source	Machine Code		Cycles
Format	Opcode	Operand	
JACC address	Exh	address byte	7

## JC

## Jump if Carry

**Syntax:** JC address

**Operation:** if CF=1, then  $pc \leftarrow k$

**Description:** If the carry flag is set, then jump to the address (place the address in the program counter).

**Condition Flags:**

CF: Carry flag unaffected.

ZF: Zero flag unaffected.

Source	Machine Code		Cycles
Format	Opcode	Operand	
JC address	Cxh	address byte	5 (4)

## JMP

## Jump

**Syntax:** JMP address

**Operation:**  $pc \leftarrow k$

**Description:** Jump unconditionally to the address (place the address in the program counter).

**Condition Flags:**

CF: Carry flag unaffected.

ZF: Zero flag unaffected.

Source	Machine Code		Cycles
Format	Opcode	Operand	
JMP address	8xh	address byte	5

## JNC

## Jump if No Carry

**Syntax:** JNC address

**Operation:** if CF=0 then  $pc \leftarrow k$

**Description:** If the carry flag is not set, then jump to the address (place the address in the program counter).

**Condition Flags:**

CF: Carry flag unaffected.

ZF: Zero flag unaffected.

Source	Machine Code		Cycles
Format	Opcode	Operand	
JNC address	Dxh	address byte	5 (4)

**JNZ****Jump if Not Zero****Syntax:** **JNZ** address**Operation:** if ZF=0 then pc ← k**Description:** If the zero flag is not set then jump to the address (place the address in the program counter).**Condition Flags:**

CF: Carry flag unaffected.

ZF: Zero flag unaffected.

Source	Machine Code		Cycles
Format	Opcode	Operand	
JNZ address	Bxh	address byte	5 (4)

**JZ****Jump if Zero****Syntax:** **JZ** address**Operation:** if ZF=1 then pc ← k**Description:** If the zero flag is set then jump to the address (place the address in the program counter).**Condition Flags:**

CF: Carry flag unaffected.

ZF: Zero flag unaffected.

Source	Machine Code		Cycles
Format	Opcode	Operand	
JZ address	Axh	address byte	5 (4)

# MOV

# Move

**Syntax:**

**MOV A, expr**  
**MOV A, [expr]**  
**MOV A, [X + expr]**  
**MOV [expr], A**  
**MOV [X + expr], A**  
**MOV X, expr**  
**MOV X, [expr]**  
**MOV X, A**  
**MOV A,X**  
**MOV PSP,A**

**Operation:**

$acc \leftarrow k$   
 $acc \leftarrow [k]$   
 $acc \leftarrow [X + k]$   
 $[k] \leftarrow acc$   
 $[X + k] \leftarrow acc$   
 $X \leftarrow k$   
 $X \leftarrow [k]$   
 $X \leftarrow acc$   
 $acc \leftarrow X$   
 $PSP \leftarrow acc$

B CPU  
 B CPU  
 B CPU

**Description:** This instruction allows for a number of combinations of moves. Immediate, direct and indexed addressing is supported. All moves involve either the accumulator or the X register.

**Condition Flags:**

CF: Carry flag unaffected.  
 ZF: Zero flag unaffected.

Source	Machine Code		Cycles
Format	Opcode	Operand	
MOV A, expr	19h	Immediate byte	4
MOV A, [expr]	1Ah	Direct address byte	5
MOV A, [X+expr]	1Bh	Offset byte	6
MOV [expr], A	31h	Direct address byte	5
MOV [X+ expr],A	32h	Offset byte	6
MOV X, expr	1Ch	Immediate byte	4
MOV X, [expr]	1Dh	Direct address byte	5
MOV A,X	40h		4
MOV X,A	41h		4
MOV PSP,A	60h		4

## NOP

## No Operation

**Syntax:**        **NOP**

**Operation:**    none

**Description:**   This one byte instruction performs no operation.

**Condition Flags:**

CF:     Carry flag unaffected  
ZF:     Zero flag unaffected.

Source	Machine Code		Cycles
Format	Opcode	Operand	
NOP	20h		4

## OR

## Bitwise OR

**Syntax:**

**OR A, expr**

**OR A, [expr]**

**OR A, [X + expr]**

**OR [expr], A**

**OR [X+ expr], A**

**Operation:**

$acc \leftarrow acc \vee k$

$acc \leftarrow acc \vee [k]$

$acc \leftarrow acc \vee [X + k]$

$[k] \leftarrow acc \vee [k]$

$[X + k] \leftarrow acc \vee [X + k]$

**Description:**   A bitwise OR of a value; k, [k] or [X+ k] and the contents of the accumulator. The result is placed in either the accumulator, [k] or [X+ k] according to the field just to the right of the opcode.

**Condition Flags:**

CF:     Always cleared.

ZF:     Set if the result is zero; cleared otherwise.

Source	Machine Code		Cycles
Format	Opcode	Operand	
OR A, expr	0Dh	Immediate byte	4
OR A, [expr]	0Eh	Direct address byte	6
OR A, [X+expr]	0Fh	Offset byte	7
OR [expr], A	33h	Direct address byte	7
OR [X+ expr], A	34h	Offset byte	8

## POP

## Pop Data Stack into Register

### Syntax

**POP A**

**POP X**

### Operation

acc ← [dsp]  
 dsp ← dsp + 1  
 X ← [dsp]  
 dsp ← dsp + 1

**Description:** Place the contents of the top of the stack into the designated register. Increment the data stack pointer.

### **Condition Flags:**

CF: Carry flag unaffected.  
 ZF: Zero flag unaffected.

Source	Machine Code		Cycles
Format	Opcode	Operand	
POP A	2Bh		4
POP X	2Ch		4

## PUSH

## Push Register into Data Stack

### Syntax

**PUSH A**

**PUSH X**

### Operation

dsp ← dsp - 1  
 [dsp] ← acc  
 dsp ← dsp - 1  
 [dsp] ← X

**Description:** Decrement the data stack pointer. Push the contents of the designated register onto the data stack.

### **Condition Flags:**

CF: Carry flag unaffected.  
 ZF: Zero flag unaffected.

Source	Machine Code		Cycles
Format	Opcode	Operand	
PUSH A	2Dh		5
PUSH X	2Eh		5

## RET

## Return

**Syntax:**         **RET**

**Operation:**      $psp \leftarrow psp - 2$   
 $pc \leftarrow [psp]$

**Description:**   Pop two bytes off of the program stack into the program counter.

**Condition Flags:** Depends on A or B versions CPUs

**A version:**

CF:     Carry restored to the value that was pushed onto the program stack.

ZF:     Zero restored to the value that was pushed onto the program stack.

**B version:**

CF:     Carry unchanged by this instruction.

ZF:     Zero unchanged by this instruction.

Source Format	Opcode	Machine Code		Cycles
		Operand	Comments	
RET	3FH		A: flags restored	8
RET	3FH		B: flags returned	8

## RETI

## Return from Interrupt

**Syntax:**         **RETI**                     **B CPU only.**

**Operation:**      $psp \leftarrow psp - 2$   
 $pc \leftarrow [psp]$

**Description:**   Pop two bytes off of the program stack into the program counter, and re-enables interrupts.

**Condition Flags:**

CF:     Carry restored to the value that was pushed onto the program stack.

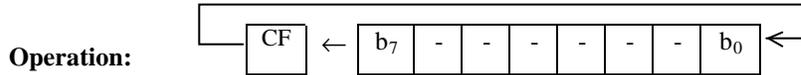
ZF:     Zero restored to the value that was pushed onto the program stack.

Source Format	Opcode	Machine Code		Cycles
		Operand	Comments	
RETI	73H		flags restored	8

## RLC

## Rotate Left Through Carry

**Syntax:** RLC A or RLC



**Description:** Shifts all bits of the accumulator one place to the left. Bit 0 is loaded with the carry flag. The most significant bit of the accumulator is loaded into the carry flag.

**Condition Flags:**

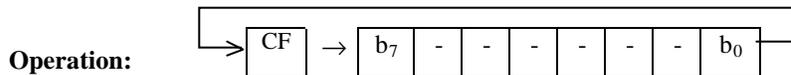
CF: Set if the MSB of the accumulator was set, before the shift, cleared otherwise.  
 ZF: Set if the result is zero; cleared otherwise.

Source	Machine Code		Cycles
Format	Opcode	Operand	
RLC A	3Dh		4

## RRC

## Rotate Right Through Carry

**Syntax:** RRC A or RRC



**Description:** Shifts all bits of the accumulator one place to the right. The carry flag is loaded into the most significant bit of the accumulator, bit 7. Bit 0 of the accumulator is loaded into the Carry flag.

**Condition Flags:**

CF: Set if LSB of the accumulator was set, before the shift, cleared otherwise.  
 ZF: Set if the result is zero; cleared otherwise.

Source	Machine Code		Cycles
Format	Opcode	Operand	
RRC A	3Eh		4

**SUB****Subtract without Borrow****Syntax:****SUB A, expr****SUB A, [expr]****SUB A, [X + expr]****Operation:** $acc \leftarrow acc - k$  $acc \leftarrow acc - [k]$  $acc \leftarrow acc - [X + k]$ 

**Description:** Subtracts a value; k, [k] or [X+ k] from the contents of the accumulator and places the result in the accumulator.

**Condition Flags:**

CF: Set if, treating the numbers as unsigned, the result &lt; 0; cleared otherwise.

ZF: Set if the result is zero; cleared otherwise.

Source Format	Machine Code		Cycles
	Opcode	Operand	
SUB A, expr	07h	Immediate byte	4
SUB A, [expr]	08h	Direct address byte	6
SUB A, [X+expr]	09h	Offset byte	7

**SBB****Subtract with Borrow****Syntax:****SBB A, expr****SBB A, [expr]****SBB A, [X + expr]****Operation:** $acc \leftarrow acc - (k + cf)$  $acc \leftarrow acc - ([k] + cf)$  $acc \leftarrow acc - ([X + k] + cf)$ 

**Description:** Subtracts a value; k, [k] or [X+ k], plus the carry flag, from the contents of the accumulator and places the result in the accumulator.

**Condition Flags:**

CF: Set if, treating the numbers as unsigned, the result &lt; 0; cleared otherwise.

ZF: Set if the result is zero; cleared otherwise.

Source Format	Machine Code		Cycles
	Opcode	Operand	
SBB A, expr	0Ah	Immediate byte	4
SBB A, [expr]	0Bh	Direct address byte	6
SBB A, [X+expr]	0Ch	Offset byte	7

## SWAP

## Swap

**Syntax:**  
**SWAP A, X**

**Operation:**

$t \leftarrow X$

$X \leftarrow \text{acc}$

$\text{acc} \leftarrow t$

**SWAP A, DSP**

$t \leftarrow \text{DSP}$

$\text{DSP} \leftarrow \text{acc}$

$\text{acc} \leftarrow t$

**Description:** Operates on either the X register or the data stack pointer. Use the temporary register to facilitate a swap of the contents of the accumulator with that of the X register or the data stack pointer.

**Condition Flags:**

CF: Carry flag unaffected.

ZF: Zero flag unaffected.

Source Format	Machine Code		Cycles
	Opcode	Operand	
SWAP A, X	2Fh		5
SWAP A, DSP	30h		5

## XOR

## Bitwise XOR

**Syntax:**  
**XOR A, expr**  
**XOR A, [expr]**  
**XOR A, [X + expr]**  
**XOR [expr], A**  
**XOR [X+ expr], A**

**Operation:**

$\text{acc} \leftarrow \text{acc} \oplus k$

$\text{acc} \leftarrow \text{acc} \oplus [k]$

$\text{acc} \leftarrow \text{acc} \oplus [X + k]$

$[k] \leftarrow \text{acc} \oplus [k]$

$[X + k] \leftarrow \text{acc} \oplus [X + k]$

**Description:** A bitwise Exclusive OR of a value; k, [k] or [X+ k] and the contents of the accumulator. The result is placed in either the accumulator, [k] or [X+ k] according to the field just to the right of the opcode.

**Condition Flags:**

CF: Cleared always.

ZF: Set if the result is zero; cleared otherwise.

Source Format	Machine Code		Cycles
	Opcode	Operand	
XOR A, expr	13h	Immediate byte	4
XOR A, [expr]	14h	Direct address byte	6
XOR A, [X+expr]	15h	Offset byte	7
XOR [expr], A	37h	Direct address byte	7
XOR [X+ expr], A	38h	Offset byte	8

## XPAGE

## Memory Page

**Syntax:**        **XPAGE**

**Operation:**    pch ← pch + 1

**Description:**   Increment the upper byte of the program counter.

**Condition Flags:**

CF:     Carry flag unaffected.

ZF:     Zero flag unaffected.

Source	Machine Code		Cycles
Format	Opcode	Operand	
XPAGE	1Fh		4

## Alphabetic Instruction Table

Source Format	Machine Code		Cycles	Bytes	Flags		
	Opcode	Operand			cf	zf	int
ADD A, expr	01h	Immediate byte	4	2	C	C	
ADD A, [expr]	02h	Direct address byte	6	2	C	C	
ADD A, [X+expr]	03h	Offset byte	7	2	C	C	
ADC A, expr	04h	Immediate byte	4	2	C	C	
ADC A, [expr]	05h	Direct address byte	6	2	C	C	
ADC A, [X+expr]	06h	Offset byte	7	2	C	C	
AND A, expr	10h	Immediate byte	4	2	0	C	
AND A, [expr]	11h	Direct address byte	6	2	0	C	
AND A, [X+expr]	12h	Offset byte	7	2	0	C	
AND [expr], A	35h	Direct address byte	7	2	0	C	
AND [X+ expr], A	36h	Offset byte	8	2	0	C	
ASL or ASL A	3Bh		4	1	C	C	
ASR or ASR A	3Ch		4	1	C	C	
CALL addr	9xh	address byte	10	2			
<b>CALL addr</b>	<b>5xh</b>	<b>address byte</b>	<b>10</b>	<b>2</b>			
CMP A, expr	16h	Immediate byte	5	2	C	C	
CMP A, [expr]	17h	Direct address byte	7	2	C	C	
CMP A, [X+expr]	18h	Offset byte	8	2	C	C	
CPL or CPL A	3Ah		4	1	1	C	
DEC A	25h		4	1	C	C	
DEC X	26h		4	1	C	C	
DEC [expr]	27h	direct address byte	7	2	C	C	
DEC [X+ expr]	28h	offset address byte	8	2	C	C	
<b>DI</b>	<b>70h</b>		<b>4</b>	<b>1</b>			<b>0</b>
<b>EI</b>	<b>72h</b>		<b>4</b>	<b>1</b>			<b>1</b>
HALT	00h		7	1			
INC A	21h		4	1	C	C	
INC X	22h		4	1	C	C	
INC [expr]	23h	direct address byte	7	2	C	C	
INC [X+ expr]	24h	offset address byte	8	2	C	C	
INDEX address	Fxh	address byte	14	2	C	C	
IORD address	29h	address byte	5	2			
IOWR address	2Ah	address byte	5	2			
IOWX [X+ expr]	39h	offset address byte	6	2			
<b>IPRET</b>	<b>1Eh</b>	<b>IO address</b>	<b>13</b>	<b>2</b>			
JACC address	Exh	address byte	7	2	C	C	
JC address	Cxh	address byte	5 (4)	2			
JMP address	8xh	address byte	5	2			
JNC address	Dxh	address byte	5 (4)	2			
JNZ address	Bxh	address byte	5 (4)	2			
JZ address	Axh	address byte	5 (4)	2			
MOV A, expr	19h	Immediate byte	4	2			
MOV A, [expr]	1Ah	Direct address byte	5	2			
MOV A, [X+expr]	1Bh	Offset byte	6	2			
MOV [expr], A	31h	Direct address byte	5	2			

## Alphabetic Instruction Table (Continued)

Source Format	Machine Code		Cycles	Bytes	Flags		
	Opcode	Operand			cf	zf	int
MOV [X+ expr],A	32h	Offset byte	6	2			
MOV X, expr	1Ch	Immediate byte	4	2			
MOV X, [expr]	1Dh	Direct address byte	5	2			
<b>MOV A,X</b>	<b>40h</b>		<b>4</b>	<b>1</b>			
<b>MOV X,A</b>	<b>41h</b>		<b>4</b>	<b>1</b>			
<b>MOV PSP,A</b>	<b>60h</b>		<b>4</b>	<b>1</b>			
NOP	20h		4	1			
OR A, expr	0Dh	Immediate byte	4	2	0	C	
OR A, [expr]	0Eh	Direct address byte	6	2	0	C	
OR A, [X+expr]	0Fh	Offset byte	7	2	0	C	
OR [expr], A	33h	Direct address byte	7	2	0	C	
OR [X+ expr], A	34h	Offset byte	8	2	0	C	
POP A	2Bh		4	1			
POP X	2Ch		4	1			
PUSH A	2Dh		5	1			
PUSH X	2Eh		5	1			
RET (CPU A)	3Fh		8	1	C	C	
RET (CPU B)	3Fh		8	1			
<b>RETI</b>	<b>73h</b>		<b>8</b>	<b>1</b>	<b>C</b>	<b>C</b>	<b>1</b>
RLC or RLC A	3Dh		4	1	C	C	
RRC or RRC A	3Eh		4	1	C	C	
SUB A, expr	07h	Immediate byte	4	2	C	C	
SUB A, [expr]	08h	Direct address byte	6	2	C	C	
SUB A, [X+expr]	09h	Offset byte	7	2	C	C	
SBB A, expr	0Ah	Immediate byte	4	2	C	C	
SBB A, [expr]	0Bh	Direct address byte	6	2	C	C	
SBB A, [X+expr]	0Ch	Offset byte	7	2	C	C	
SWAP A, X	2Fh		5	1			
SWAP A, DSP	30h		5	1			
XOR A, expr	13h	Immediate byte	4	2	0	C	
XOR A, [expr]	14h	Direct address byte	6	2	0	C	
XOR A, [X+expr]	15h	Offset byte	7	2	0	C	
XOR [expr], A	37h	Direct address byte	7	2	0	C	
XOR [X+ expr], A	38h	Offset byte	8	2	0	C	
XPAGE	1Fh		4	1			

Instructions in bold are not available for both CPU types

cf: carry flag, zf: zero flag, int: B CPU interrupt enable

In Flag columns blank is unchanged, C is changed, and 0 or 1 is set to that value

The conditional jump instructions (JC, JNC, JZ, JNZ) list two numbers in the Cycles column. The first number is the number of cycles for the instruction execution when the branch is taken. The second number, shown in parenthesis, is the number of cycles when the branch is not taken.

### CPU Product Identification Table

Product ID	CPU type	RAM	EPROM
63000	63 / A CPU	128 bytes	2K bytes
63001	63 / A CPU	128 bytes	4K bytes
63100	63 / A CPU	128 bytes	2K bytes
63101	63 / A CPU	128 bytes	4K bytes
63200	63 / A CPU	128 bytes	2K bytes
63201	63 / A CPU	128 bytes	4K bytes
63221	64 / B CPU	96 bytes	3K bytes
63231	64 / B CPU	96 bytes	3K bytes
63411	64 / B CPU	256 bytes	4K bytes
63412	64 / B CPU	256 bytes	6K bytes
63413	64 / B CPU	256 bytes	8K bytes
63511	64 / B CPU	256 bytes	4K bytes
63512	64 / B CPU	256 bytes	6K bytes
63513	64 / B CPU	256 bytes	8K bytes
63612	64 / B CPU	256 bytes	6K bytes
63613	64 / B CPU	256 bytes	8K bytes
63722	64 / B CPU	256 bytes	6K bytes
63723	64 / B CPU	256 bytes	8Kbytes
63742	64 / B CPU	256 bytes	6K bytes
63743	64 / B CPU	256 bytes	8K bytes
64013	64 / B CPU	256 bytes	8K bytes
64113	64 / B CPU	256 bytes	8K bytes
65013	64 / B CPU	256 bytes	8K bytes
65113	64 / B CPU	256 bytes	8K bytes
66013	64 / B CPU	256 bytes	8K bytes
66113	64 / B CPU	256 bytes	8Kbytes

