
PSoC™ Designer: Assembly Language

**User Guide
Revision 1.07**

CMS10003A
Last Revised: May 30, 2001
Cypress MicroSystems, Inc.

Copyright Information

Copyright © 2000-2001 Cypress Microsystems, Inc. All rights reserved.

Programmable System on Chip: PSoC™ is a trademark of Cypress Microsystems.

Copyright © 1999-2000 ImageCraft Creations Inc. All rights reserved.

The information contained herein is subject to change without notice.

Two-Minute Overview

This two-minute overview of *PSoC Designer: Assembly Language User Guide* was purposefully placed up front for you advanced engineers who are ready to program the chip but need a *quick* point in the right direction. (Now we only have a minute and-a-half left.)

Overview 35 seconds You have the chip, configured the device, and placed the User Modules in the PSoC blocks, now you are ready to program the chip using assembly language code.

This guide provides:

- instructions related to address spaces, modes, and destination of results
- explanation of assembly-file syntax, input and output
- assembler directives
- the complete instruction set.

Basics 15 seconds Upon opening PSoC Designer, click the Application Editor icon in the toolbar to access the Assembler and pre-configured source files.

The source files appear in the left frame. Double-click individual files to appear in the main frame where you can add and modify code using the enabled edit icons.

Quick Reference 30 seconds Click a hyperlink to reference key material:

[Notation Standards](#)

[Microprocessor and related address information](#)

[Assembly File Syntax](#)

[List File Format](#)

[Assembler Directives](#)

[Instruction Set or Instruction Set Reference Table](#)

Bottom Line 10 seconds Programmable System on Chip PSoC™ Designer empowers you to customize the functionality you desire into the M8C microprocessor.



Time's up... Now get to work.

Documentation Conventions

Following, are easily identifiable conventions used throughout the PSoC Designer suite of product documentation.

Convention	Usage
Times New Roman Size 10-12	Displays an input command: <code>iasm8c -g</code>
Courier New Size 10	Displays output: <pre> // Created by PSoC Designer // from template BOOT.ASM // Boot Code, from Reset // // Change this file at your own risk! --- 000AREA TOP(ABS) org 0 0000 8033 jmp __start 0002 8031 jmp __start 0004 801F jmp Interrupt0 0006 801E jmp Interrupt1 </pre>
Courier Size 12	Displays file locations: <code>...\Project Name\output</code>
Arial Size 8	Displays Instruction Set Reference Table data: <code>01h ADD A k</code>
<i>Italics</i>	Displays file names: <i>projectname.rom</i>
[Ctrl] [C]	Displays keyboard commands: [Enter]
File >> Open	Displays menu paths: Edit >> Cut

Notation Standards

Following, is input notation referenced throughout this guide and wherever applicable in the PSoC Designer suite of product documentation.

Internal Registers:

Notation	Description
A	Primary Accumulator
CF	Carry Flag
expr	Expression
F	Flags (ZF, CF, and Others)
I	Operand 1 Value
K	Operand 2 Value
PC	(PCH,PCL)
SP	Stack Pointer
X	X Register
ZF	Zero Flag

Assembler Directives:

Symbol	Assembler Directive
AREA	Area
BLK	RAM Block (in Bytes)
BLKW	RAM Block in Words (16 Bits)
DB	Define Byte
DS	Define ASCII String
DSU	Define UNICODE String
DW	Define Word (2 Bytes)
DWL	Define Word with Little Endian Ordering
ELSE	Alternative Result of IF...ELSE...ENDIF
ENDIF	End of IF...ELSE...ENDIF
EQU	Equate Label to Variable Value
EXPORT	Export
IF	Conditional Assembly
INCLUDE	Include Source File
MACRO/ENDM	Macro Definition Start/End
ORG	Area Origin

Assembly Syntax Expressions:

Precedence	Expression	Symbol	Form
1	Bitwise Complement	~	(~ a)
2	Multiplication	*	(a * b)
	Division	/	(a / b)
3	Addition	+	(a + b)
	Subtraction	-	(a - b)
4	Bitwise AND	&	(a & b)
5	Bitwise XOR	^	(a ^ b)
6	Bitwise OR		(a b)
7	High Byte of an Address	>	(>a)
8	Low Byte of an Address	<	(<a)

Only the Addition expression (+) may apply to a relocatable symbol (i.e., an external symbol). All other expressions must be applied to constants or symbols resolvable by the assembler (i.e., a symbol defined in the file).

If a dot (.) appears in an expression, then the current value of the Program Counter (PC) is used in place of the dot.

Table of Contents

Two-Minute Overview	2
<i>Quick-start summary for advanced users who are ready to dive in.</i>	
Documentation Conventions	3
<i>Lists conventions used in this guide and throughout the PSoC Designer suite.</i>	
Notation Standards	4
<i>Lists notation referenced in this guide and throughout the PSoC Designer suite.</i>	
Section 1. Introduction	9
<i>Describes purpose of user guide, overviews sections, and summarizes product information.</i>	
1.1. <u>Purpose</u>	9
1.2. <u>Section Overview</u>	9
1.3. <u>Product Updates</u>	10
1.4. <u>Support</u>	10
Section 2. Accessing the Assembler	11
<i>Describes how to access the Assembler and use its features.</i>	
2.1. <u>Opening PSoC Designer</u>	11
2.2. <u>Accessing the Assembler</u>	11
2.3. <u>Menu Options</u>	12
Section 3. The Microprocessor	13
<i>Discusses the microprocessor and setting instructions.</i>	
3.1. <u>Address Spaces</u>	14
3.2. <u>Instruction Format</u>	15
3.3. <u>Addressing Modes</u>	16
3.4. <u>Destination of Instruction Results</u>	22
Section 4. Assembly File Syntax	23
<i>Discusses assembly source-file syntax.</i>	
4.1. <u>Syntax Details</u>	23
4.2. <u>Syntax</u>	23
Section 5. List File Format	27
<i>Displays and explains assembly-file formats.</i>	

Section 6. Assembler Directives 29

Lists and defines all assembler directives.

6.1.	<u>Area</u>	29
6.2.	<u>RAM Block</u>	31
6.3.	<u>RAM Block in Words</u>	31
6.4.	<u>Define Byte</u>	32
6.5.	<u>Define ASCII String</u>	32
6.6.	<u>Define UNICODE String</u>	33
6.7.	<u>Define Word</u>	33
6.8.	<u>Define Word, Little Endian Ordering</u>	34
6.9.	<u>Alternative Result of IF...ELSE...ENDIF</u>	34
6.10.	<u>IF...ELSE...ENDIF - ENDIF</u>	34
6.11.	<u>Equate Label</u>	35
6.12.	<u>Export</u>	35
6.13.	<u>IF...ELSE...ENDIF - IF</u>	35
6.14.	<u>Include Source File</u>	36
6.15.	<u>Macro Definition Start/Macro Definition End</u>	36
6.16.	<u>Area Origin</u>	37

Section 7. Instruction Set..... 39

Describes notation for the instruction set.

7.1.	<u>Add with Carry</u>	40
7.2.	<u>Add without Carry</u>	41
7.3.	<u>Bitwise AND</u>	42
7.4.	<u>Arithmetic Shift Left</u>	43
7.5.	<u>Arithmetic Shift Right</u>	43
7.6.	<u>Call Function</u>	44
7.7.	<u>Non-destructive Compare</u>	45
7.8.	<u>Complement Accumulator</u>	45
7.9.	<u>Decrement</u>	46
7.10.	<u>Halt</u>	47
7.11.	<u>Increment</u>	48
7.12.	<u>Table Read INDEX</u>	49
7.13.	<u>Jump Accumulator</u>	50
7.14.	<u>Jump if Carry</u>	51
7.15.	<u>Jump</u>	52
7.16.	<u>Jump if No Carry</u>	53
7.17.	<u>Jump if Not Zero</u>	54
7.18.	<u>Jump if Zero</u>	55
7.19.	<u>Long Call</u>	56
7.20.	<u>Long Jump</u>	56
7.21.	<u>Move</u>	57
7.22.	<u>Move Indirect, Post-Increment to Memory</u>	58
7.23.	<u>No Operation</u>	59
7.24.	<u>Bitwise OR</u>	60
7.25.	<u>Pop Stack into Register</u>	61
7.26.	<u>Push Register onto Stack</u>	61
7.27.	<u>Return</u>	62

7.28.	<u>Return from Interrupt</u>	62
7.29.	<u>Rotate Left through Carry</u>	63
7.30.	<u>Table Read ROMX</u>	63
7.31.	<u>Rotate Right through Carry</u>	64
7.32.	<u>Subtract with Borrow</u>	65
7.33.	<u>Subtract without Borrow</u>	66
7.34.	<u>Swap</u>	67
7.35.	<u>System Supervisory Call</u>	68
7.36.	<u>Test with Mask</u>	69
7.37.	<u>Bitwise XOR</u>	70
<u>Section 8. Compile/Assemble Error Messages</u>		71
<i>Lists all PSoC Designer compile/assemble errors and warnings.</i>		
8.1.	<u>Preprocessor</u>	71
8.2.	<u>C Compiler</u>	72
8.3.	<u>Assembler</u>	75
8.4.	<u>Linker</u>	76
<u>Instruction Set Reference Table</u>		77
<i>Supplies instruction set in a quick-reference table.</i>		
<u>Appendix A: Application Interface Notes</u>		78
<i>Discusses interfacing between assembly language and 'C' in PSoC Designer.</i>		
<u>Index</u>		79

Section 1. Introduction

1.1. Purpose

The *PSoC Designer: Assembly Language User Guide* will guide you through the process of programming the M8C microprocessor in the assembly language.

1.2. Section Overview

Following, is a brief description of each section in this user guide:

Section 1. Introduction

This section describes the purpose of this guide, overviews each section, and gives product upgrade and support information.

Section 2. Accessing the Assembler

This section describes how to open PSoC Designer and access the Assembler. It also defines applicable menu options.

Section 3. The Microprocessor

This section discusses the M8C and explains address spaces, instruction format, and destination of instruction results. It also lists all the addressing modes with examples.

Section 4. Assembly File Syntax

This section provides assembly-language-source syntax including labels, mnemonics, operands, expressions, and comments.

Section 5. List File Format

This section displays a small assembly program with its listing file generated from the assembly-source input. An explanation for the file contents is also provided.

Section 6. Assembler Directives

This section lists and describes, with examples, all active assembler directives. These directives communicate commands to the assembler program.

Section 7. Instruction Set

This section provides a detailed list of all M8C instructions.

Section 8. Compile/Assemble Error Messages

This section provides several lists of compile/assemble and related errors and warnings.

1.3. Product Upgrades

Cypress MicroSystems provides scheduled upgrades and version enhancements for PSoC Designer *free of charge*. You can order the upgrades from your distributor on CD-ROM or, better yet, download them directly from the Cypress MicroSystems web site at <http://www.cypressmicro.com/>.

Also provided at the web site are critical updates to system documentation. To stay current with system functionality you can find documentation updates under the Documentation hyperlink, again, at <http://www.cypressmicro.com/>.

Check the Cypress MicroSystems web site frequently for both product and documentation updates. As the M8C and PSoC Designer evolve, you can be sure that new features and enhancements will be added. To register and receive product update notification go to <http://www.cypressmicro.com/registerme/>.

1.4. Support

Support for the Assembler is *free*. For details, see the *PSoC Designer: Integrated Development Environment User Guide*.

Section 2. Accessing the Assembler

In this section you will learn to quickly access the Assembler and work its functionality from within PSoC Designer.

It is assumed that PSoC Designer is installed and up and running on your computer. It is also assumed that a project has been created and the device configured. If not, see the *PSoC Designer: Integrated Development Environment User Guide*.

2.1. Opening PSoC Designer

To open PSoC Designer go to:

Start >> Programs >> Cypress MicroSystems >> PSoC Designer.

Note that upon opening the system, that last project you were in will be loaded as the default.

2.2. Accessing the Assembler

The Assembler is an application accessed and run as a batch process from within PSoC Designer, much like the C Compiler. It operates on assembly-language source, constructed by you, to produce executable code. This code is then built into a single executable file that can be downloaded into the In-Circuit Emulator (ICE), where the functionality of the microprocessor can be emulated and debugged.



To access assembly-language source, click the **Application Editor** icon in the toolbar.

The project source files appear in the left frame (source tree). Double-click individual files to appear in the main active window where you can add and modify code using the enabled edit icons.



To compile the source files for the current project, click the **Compile/Assemble** icon in the toolbar.





To build the current project, click the **Build** icon in the toolbar.

These combined actions construct the entire project by assembling.asm files and compiling.c files from the `c:\project name\` directory. The intermediate files generated are then placed in the `c:\project name\obj` directory as .o and .lis files. Building (linking) the intermediate files generates the program files and places them in `c:\project name\output` directory as *projectname.dbg*, *projectname.hex*, *projectname.mp*, *projectname.rom*, and *projectname.lst*.

2.3. Menu Options

Following, is a description of the menu options available for use with the Assembler:

Icon	Menu	Shortcut	Feature
	Compile/Assemble	[Ctrl] [F7]	Compiles/assembles the most prominent open, active file (.c or .asm)
	Build	[F7]	Builds entire project and links applicable files

Section 3. The Microprocessor

In this section you will learn about the M8C, address spaces, instruction format, and destination of instruction results. You can also view and reference all the addressing modes with examples.

The M8C is an enhanced 8-bit microprocessor core. It supports 8-bit operations and has been optimized to be small and fast.

The Internal registers are: the accumulator 'A'; the 'F' flag register; the index register 'X'; the stack pointer 'SP'; and the program counter 'PC'. All registers are 8 bits wide except 'PC' which is composed of two 8-bit registers (PCH and PCL) which together form a 16-bit register. The M8C supports a full 16-bit address to program memory. Following is a look at PCH and PCL:

PCH								PCL							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a

Upon reset, A, X, PC, and SP are reset to 0x00. The SP grows up, with a post-increment, and always points to the next free byte for pushing. The flag ('F') register is cleared except for the Z status bit.

F							
7	6	5	4	3	2	1	0
XA[2]	XA[1]	XA[0]	XIO	--	C	Z	IE

The flag register has three status bits: Global Interrupt Enable bit 0; Zero flag bit 1; Carry Flag bit 2; and three extended RAM address bits XA[2:0] at 7:5, and the extended IO address at bit 4. Bit 3 is the supervisory state and will remain cleared. The flags are affected by arithmetic, logic, and shift operations. The manner in which each flag is changed is dependent upon the instruction being executed. [Section 7. Instruction Set](#) includes information about how each instruction affects the flags.

All instructions are 1, 2, or 3 bytes wide and fetched from program memory, in a separate address space from data memory. The first byte of an instruction is an 8-bit constant, referred to as the Opcode. Depending on the instruction, there can be one or two succeeding bytes that encode address or operand information.

3.1. Address Spaces

There are three separate address spaces implemented in the Assembler: Register space (REG), data RAM space, and program memory space.

The Register space is accessed through the MOV and LOGICAL instructions. There are 8 address bits available to access the Register space, plus an extended address bit via the flag register bit 4.

The data RAM space contains the data/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 space. Data RAM addresses are 8 bits wide, although for RAM sizes 128 bytes or smaller, not all bits are used. The Extended Address flag bits (XA[2:0]) are used to address beyond the first 256 bytes of RAM. Depending on the memory size implemented on a particular device, any or all of the Extended Address bits may not be implemented. These 3 bits provide an 11-bit RAM address for addressing up to 2 kilobytes as 8 pages of 256 bytes each. The flag register must be manipulated to change RAM page addresses.

All stack operations force XA[2:0] on the bus to be zero (leaving flag values intact) so that the stack is constrained to the first 256 bytes page.

The program memory space 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 M8C automatically advances PCH when a page boundary needs to be crossed. The user need not be concerned with program memory page boundaries, as they are invisible within the programming module. The one exception to this is that non-jump instructions ending on a page boundary will take an extra cycle to complete. Jump instructions are not affected in this manner.

The INDEX instruction is used to move information in table form from the program memory space into the accumulator. It has one operand, which is the lower part of the base address of a program memory table. The lower nibble of the INDEX opcode forms the upper part of the base address. This yields a 12-bit-twos-complement-relative-address that when added to the PC, yields the base address of the table. 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:

```
tab1:  DS "hello"      ;define a table called tab1
        MOV A, 04
        INDEX tab1   ;fetch the 5th byte ("o") from table tab1.
```

The ROMX instruction will also move information from the program memory space into the accumulator.

There are several types of jump instructions to control program flow. The Long Jump (LJMP) instruction has two operands that together form a 16-bit absolute address. All other jump instructions have 12-bit-tvos-complement-relative-addresses that are added to the PC to form the jump target. The lower 8 bits of the relative address are contained in the operand, and the upper 4 bits are the lower nibble of the opcode.

As you will see, the Long Call (LCALL) instruction is the same as the LJMP instruction in that it too has two operands that together form a 16-bit absolute address. The other call instruction has a 12-bit-tvos-complement-relative-address that is added to the PC to form the jump target. The lower 8 bits of the relative address are contained in the operand, and the upper 4 bits are the lower nibble of the opcode. For further reference, see [Section 7. Instruction Set](#).

3.2. Instruction Format

Instruction addressing is divided into two groups: (1) Logic, arithmetic, and data movement functions (unconditional); (2) jump and call instructions, including INDEX (conditional).

In the following descriptions “0” indicates unconditional instruction bit, “1” indicates conditional instruction bit, “a” indicates bits used to store an address, “d” indicates bits used to store a data value, and “i” indicates indeterminate data (as related to instruction format).

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

Single-Byte Instruction:

Instruction							
7	6	5	4	3	2	1	0
0	i	i	i	i	i	i	i

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	i	i	i	i	i	i	i	a/d	a/d	a/d	a/d	a/d	a/d	a/d	a/d

Triple-Byte Instruction:

Instruction Byte								Address Byte								Address or Data Byte							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	i	i	i	i	i	i	i	a	a	a	a	a	a	a	a	a/d	a/d	a/d	a/d	a/d	a/d	a/d	a/d

Most jumps, plus CALL and INDEX, 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 kilobytes, a three-byte format is used in Big Endian format. Following, is the format for these short instructions:

Instruction Byte								Address Byte							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
1	i	i	i	a	a	a	a	a	a	a	a	a	a	a	a

The long CALL and JUMP have the following format:

Instruction Byte								MS Address Byte								LS Address Byte							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
1	i	i	i	i	i	i	i	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a

3.3. Addressing Modes

Ten addressing modes are supported; Source Immediate, Source Direct, Source Indexed, Destination Direct, Destination Indexed, Destination Direct Immediate, Destination Indexed Immediate, Destination Direct Direct, Source Indirect Post Increment, and Destination Indirect Post Increment.

The address mode is inferred from the syntax of the assembly code. The square brackets [] are used to denote one level of indirection. The active addressing modes are illustrated in the following examples:

3.3.1. Source Immediate

The result of an instruction using this addressing mode is placed in the A register, F register, or X register, which is specified as part of the instruction opcode. Operand 1 is an immediate value that serves as a source for the instruction. Arithmetic instructions require two sources, the second source is the A register or X register specified in the opcode. Instructions using this addressing mode are two bytes in length.

Opcode	Operand 1
Instruction	Immediate Value

Example:

ADD	A,	7	; In this case, the immediate value of 7 is added with the Accumulator, and the result is placed in the Accumulator.
MOV	X,	8	; In this case, the immediate value of 8 is moved to the X register.
AND	F,	9	; In this case, the immediate value of 8 is logically Anded with the F register and the result is placed in the F register.

3.3.2. Source Direct

The result of an instruction using this addressing mode is placed in either the A register or the X register, which is specified as part of the instruction opcode. Operand 1 is an address that points to a location in either the RAM memory space or the Register space that is the source for the instruction. Arithmetic instructions require two sources, the second source is the A register or X register specified in the opcode. Instructions using this addressing mode are two bytes in length.

Opcode	Operand 1
Instruction	Source Address

Example:

ADD	A,	[7]	; In this case, the value in the memory location at address 7 is added with the Accumulator, and the result is placed in the Accumulator.
MOV	X,	REG[8]	; In this case, the value in the Register space at address 8 is moved to the X register.

3.3.3. Source Indexed

The result of an instruction using this addressing mode is placed in either the A register or the X register, which is specified as part of the instruction opcode. Operand 1 is added to the X register forming an address that points to a location in either the RAM memory space or the Register space that is the source for the instruction. Arithmetic instructions require two sources, the second source is the A register or X register specified in the opcode. Instructions using this addressing mode are two bytes.

Opcode	Operand 1
Instruction	Source Index

Example:

```

ADD    A,    [X+7]    ; In this case, the value in the memory location at
                    ; address X + 7 is added with the Accumulator,
                    ; and the result is placed in the Accumulator.
MOV    X,    REG[X+8] ; In this case, the value in the Register space at
                    ; address X + 8 is moved to the X register.

```

3.3.4. Destination Direct

The result of an instruction using this addressing mode is placed within either the RAM memory space or the Register space. Operand 1 is an address that points to the location of the result. The source for the instruction is either the A register or the X register, which is specified as part of the instruction opcode. Arithmetic instructions require two sources, the second source is the location specified by Operand 1. Instructions using this addressing mode are two bytes in length.

Opcode	Operand 1
Instruction	Destination Address

Example:

```

ADD    [7]    A    ; In this case, the value in the memory location at
                    ; address 7 is added with the Accumulator, and the
                    ; result is placed in the memory location at address
                    ; 7. The Accumulator is unchanged.
MOV    REG[8] A    ; In this case, the Accumulator is moved to the
                    ; Register space location at address 8. The
                    ; Accumulator is unchanged.

```

3.3.5. Destination Indexed

The result of an instruction using this addressing mode is placed within the either the RAM memory space or the Register space. Operand 1 is added to the X register forming the address that points to the location of the result. The source for the instruction is either the A register or the X register, which is specified as part of the instruction opcode. Arithmetic instructions require two sources, the second source is the location specified by Operand 1 added with the X register. Instructions using this addressing mode are two bytes in length.

Opcode	Operand 1
Instruction	Destination Index

Example:

```
ADD  [X+7]  A      ; In this case, the value in the memory location at
                   ; address X+7 is added with the Accumulator, and
                   ; the result is placed in the memory location at
                   ; address X+7. The Accumulator is unchanged.
```

3.3.6. Destination Direct Immediate

The result of an instruction using this addressing mode is placed within either the RAM memory space or the Register space. Operand 1 is the address of the result. The source for the instruction is Operand 2, which is an immediate value. Arithmetic instructions require two sources, the second source is the location specified by Operand 1. Instructions using this addressing mode are three bytes in length.

Opcode	Operand 1	Operand 2
Instruction	Destination Address	Immediate Value

Example:

```
ADD  [7]      5      ; In this case, value in the memory location at
                   ; address 7 is added to the immediate value of 5,
                   ; and the result is placed in the memory location at
                   ; address 7.

MOV  REG[8]   6      ; In this case, the immediate value of 6 is moved
                   ; into the Register space location at address 8.
```

3.3.7. Destination Indexed Immediate

The result of an instruction using this addressing mode is placed within either the RAM memory space or the Register space. Operand 1 is added to the X register to form the address of the result. The source for the instruction is Operand 2, which is an immediate value. Arithmetic instructions require two sources, the second source is the location specified by Operand 1 added with the X register. Instructions using this addressing mode are three bytes in length.

Opcode	Operand 1	Operand 2
Instruction	Destination Index	Immediate Value

Example:

ADD	[X+7]	5	; In this case, the value in the memory location at address X+7 is added with the immediate value of 5, and the result is placed in the memory location at address X+7.
MOV	REG[X+8]	6	; In this case, the immediate value of 6 is moved into the location in the Register space at address X+8.

3.3.8. Destination Direct Direct

The result of an instruction using this addressing mode is placed within the RAM memory. Operand 1 is the address of the result. Operand 2 is an address that points to a location in the RAM memory that is the source for the instruction. This addressing mode is only valid on the MOV instruction. The instruction using this addressing mode is three bytes in length.

Opcode	Operand 1	Operand 2
Instruction	Destination Address	Source Address

Example:

MOV	7	8	; In this case, the value in the memory location at address 7 is moved to the memory location at address 8.
-----	---	---	---

3.3.9. Source Indirect Post Increment

The result of an instruction using this addressing mode is placed in the Accumulator. Operand 1 is an address pointing to a location within the memory space, which contains an address (the indirect address) for the source of the instruction. The indirect address is incremented as part of the instruction execution. This addressing mode is only valid on the MVI instruction. The instruction using this addressing mode is two bytes in length. See [Section 7. Instruction Set](#) for further details on MVI instruction.

Opcode	Operand 1
Instruction	Source Address Address

Example:

```
MVI    A    [8]    ; In this case, the value in the memory location at
                    ; address 8 points to a memory location that
                    ; contains an indirect address. The memory
                    ; location pointed to by the indirect address is
                    ; moved into the Accumulator. The indirect
                    ; address is then incremented.
```

3.3.10. Destination Indirect Post Increment

The result of an instruction using this addressing mode is placed within the memory space. Operand 1 is an address pointing to a location within the memory space, which contains an address (the indirect address) for the destination of the instruction. The indirect address is incremented as part of the instruction execution. The source for the instruction is the Accumulator. This addressing mode is only valid on the MVI instruction. The instruction using this addressing mode is two bytes in length.

Opcode	Operand 1
Instruction	Destination Address Address

Example:

```
MVI    [8]    A    ; In this case, the value in the memory location at
                    ; address 8 points to a memory location that
                    ; contains an indirect address. The accumulator is
                    ; moved into the memory location pointed to by the
                    ; indirect address. The indirect address is then
                    ; incremented.
```

3.4. Destination of Instruction Results

The result of a given instruction is stored in the entity, which is placed next to the opcode in the assembly code. This allows for 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 (i2 == second instruction byte, i3 == third instruction byte):

Syntax	Operation
AND A, expr	$acc \leftarrow acc \& i2$
AND A, [expr]	$acc \leftarrow acc \& [i2]$
AND A, [X+expr]	$acc \leftarrow acc \& [x + i2]$
AND [expr], A	$[i2] \leftarrow acc \& [i2]$
AND [X+expr], A	$[x + i2] \leftarrow acc \& [x + i2]$
AND [expr], expr	$[i2] \leftarrow i3 \& [i2]$
AND [X+expr], expr	$[x + i2] \leftarrow i3 \& [x + i2]$

The ordering of the entities within the instruction determines where the result of the instruction is stored.

Section 4. Assembly File Syntax

In this section you will learn applicable assembly file syntax and placement. You can also view real-life assembly language input from PSoC Designer.

4.1. Syntax Details

Assembly language instructions reside in source files with .asm extensions in the left frame of Application Editor. (See [Section 7. Instruction Set](#) for the complete set.) Each line of the source file may contain up to five keyword-types of information.

The following table gives critical details about each keyword-type:

Keyword Type	Critical Details
Label	A symbolic name followed by a colon (:).
Mnemonic	An assembly language keyword.
Operands	Follows the Mnemonic.
Expression	Is usually addressing modes with labels and should be inside parenthesis.
Comment	Can follow Operands or Expressions and start in any column if the first non-space character is either a C++ style comment (//) or semi-colon (;).

4.2. Syntax

Instructions in an assembly file have one operation on a single line. For readability, separate each keyword-type by tabbing once or twice (approximately 5-10 white spaces).

Note that in all source files, including .asm, the maximum number of characters allowed per line is 2,048. The maximum number of characters allowed per word is 256. These limits are imposed by the PSoC Designer development software.

Following are type definitions and an example of assembly-file syntax:

4.2.1. Labels

A label is a case sensitive set of alphanumeric characters and underscores (_) followed by a colon. A label will be assigned a value, but may also be used as operands.

A label is assigned the value of the current Program Counter unless it is defined on a line with an EQU directive. (See [Section 6. Assembler Directives](#) for the entire list with sample directives.) Labels can be included on any line, including blank lines. 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 period (.) character then that label has only local scope and/or existence between two global labels, i.e. labels that do not begin with a period (.) character and are exported. These local labels can re-use the same names within differing global scopes.

Local labels are restricted to use between global labels, i.e. you cannot use one before a global label has been defined.

4.2.2. Mnemonics

A mnemonic is an assembly instruction, assembler directive, or a user-defined macro name. All are defined in more detail in sections [6. Assembler Directives](#) and [7. Instruction Set](#). There can be 0 or 1 mnemonic on a line of assembly code. Mnemonics, with the exception of macro names, are case-sensitive.

4.2.3. Operands

Operands either specify the addressing mode for an instruction as described in **Addressing Modes** and **Destination of Instruction Results** of [Section 3. The Microprocessor](#), or are an expression that specifies a value used by an instruction. The number and type of operands accepted on a line depends on the mnemonic on that line. See sections [6. Assembler Directives](#) and [7. Instruction Set](#) for information on operands accepted by specific mnemonics. A line with no mnemonic must have no operands.

There are two types of operands; labels and constants. Following, is a description of each:

- **Labels:** Labels used as operands are replaced with their defined value. Definitions may be made anywhere within the source file as described in the previous information on labels. A label is defined with a colon following the name, but the colon is not part of the name. That is, when used as an operand, do not include the colon as part of the label name.

- **Constants:** Constants are specified as binary, decimal, hexadecimal, or character. The radix of a number prefixes the number. Standard radices include 0b or % for binary, 0 for octal, and 0x or \$ for hexadecimal.

Note that the Assembler will not generate an error if values following a binary prefix contain non binary values (e.g., 2..Z).

A hexadecimal number can also be specified by a number followed by h or H. If no radix is specified, the number is assumed to be decimal. For example, 0b1010, 10, 0xA, and Ah are all equivalent.

Character constants are enclosed by single quotes and have the ASCII value of the character. For example, 'A' has the value of 41h. The backslash (\) is used as an escape character. To enter a single quote (') as a character, use \'. To enter a backslash (\), use \\.

A "." is shorthand for the current value of the Program Counter. Following are a few examples:

```
MOV A, .      ; Moves to low byte of the Program Counter to A
JMP.+2       ; Jumps to 2 bytes past the current Program
              ; Counter location
```

4.2.4. Expressions

Expressions may be constructed using a number of algebraic and logical operators with either labels or constants. See the order of precedence:

Precedence	Expression	Symbol	Form
1	Bitwise Complement	~	(~ a)
2	Multiplication	*	(a * b)
	Division	/	(a / b)
3	Addition	+	(a + b)
	Subtraction	-	(a - b)
4	Bitwise AND	&	(a & b)
5	Bitwise XOR	^	(a ^ b)
6	Bitwise OR		(a b)
7	High Byte of an Address	>	(>a)
8	Low Byte of an Address	<	(<a)

Only the Addition expression (+) may apply to a relocatable symbol (i.e., an external symbol). All other expressions must be applied to constants or symbols resolvable by the assembler (i.e., a symbol defined in the file).

If a dot (.) appears in an expression, then the current value of the Program Counter (PC) is used in place of the dot.

Currently, expressions must be enclosed by parenthesis.

4.2.5. Comments

A Comment is anything following a semicolon (;) or a double slash (//) to the end of a line. It is usually used to explain the assembly code and may be placed anywhere in the source file. The Assembler ignores comments, however they are written to the listing file.

Section 5. List File Format

In this section you will view an assembly program with its output generated from the input described. You will also find an explanation of the data.

When you build a project (using all assembly files), a listing file with an .lst extension is created. The listing shows how the assembly program is mapped into a section of code beginning at address 0. The linking (building) process will resolve the final addresses. This file also provides a listing of errors and warnings, and a reference table of labels.

Below is an assembly-language excerpt (*main.asm*) of Example_ADC_28pin (PSoC Designer Example project) and its listing file (*Example_ADC_28pin.lst*):

main.asm File:

```

;*****
; Example_ADC_28pin, a PSoC Pup Board Project.
;
; Purpose:
; To demonstrate the operation of the 12-Bit Incremental Analog-to-
; Digital Converter User Module of the PSoC micro controller. A
; Programmable Gain Amplifier with unity gain is also incorporated.
;*****

include "m8c.inc"
include "PGA_1.inc"
include "ADCINC12_1.inc"

area bss(RAM)
    ADCVal:    BLK    1                ;Temp variable containing 8
                                           ;most significant bits of ADC
                                           ;result

area text(ROM,REL)

export ADCVal
export _main

_main:
    or        F,01h                ;Enable interrupts
    mov       A,ADCINC12_1_FULLPOWER ;Set power level
    call      ADCINC12_1_Start
    mov       A,00h                ;Set for continuous sampling
    call      ADCINC12_1_GetSamples
    mov       A,PGA_1_FULLPOWER
    call      PGA_1_Start

```

Example_ADC_28pin.lst File (After Project Build):

```

(0083) include "m8c.inc"
(0084) include "PGA_1.inc"
(0085) include "ADCINC12_1.inc"
(0086)
(0087) area bss(RAM)
(0088)     ADCVal:   BLK 1                               ;Temp variable
                                                    ;containing 8 most
                                                    ;significant bits of
                                                    ;ADC result

(0089) area text(ROM,REL)
(0090)
(0091) export ADCVal
(0092) export _main
(0093)
(0094) _main:
(0095)     or      F,01h                               ;Enable interrupts
    _main:
    0100: 71 01    OR      F,1
(0096)     mov    A,ADCINC12_1_FULLPOWER             ;Set power level
    0102: 50 03    MOV     A,3
(0097)     call  ADCINC12_1_Start
    0104: 91 5D    CALL   ADCINC12_1_SetPower
(0098)     mov    A,00h                               ;Set for continuous
                                                    ;sampling
    0106: 50 00    MOV     A,0
(0099)     call  ADCINC12_1_GetSamples
    0108: 91 64    CALL   ADCINC12_1_GetSamples
(0100)     mov    A,PGA_1_FULLPOWER
    010A: 50 03    MOV     A,3
(0101)     call  PGA_1_Start
    010C: 90 D3    CALL   _PGA_1_Start

```

The first column of the listing file identifies the line in the code.

Starting at “0100: ” location is the absolute address at which the corresponding instruction is stored. Example:

```
0100: 71 01    OR      F,1
```

“71” is the operand for the OR instruction and the F register. “01” (and “1”) is the value.

The last column shows the source description contained in the assembly source file.

The .lst file will ONLY be created if there are no source errors resulting from a build. If there are compile/assemble or linker errors upon a project build, the .lst file retains source from the last successful build.

Section 6. Assembler Directives

In this section you will learn all active assembler directives that can be used in the assembly language code.

The PSoC Designer Assembler allows the assembler directives listed below:

Symbol	Assembler Directive
AREA	Area
BLK	RAM Block (in Bytes)
BLKW	RAM Block in Words (16 Bits)
DB	Define Byte
DS	Define ASCII String
DSU	Define UNICODE String
DW	Define Word (2 Bytes)
DWL	Define Word with Little Endian Ordering
ELSE	Alternative Result of IF...ELSE...ENDIF
ENDIF	End of IF...ELSE...ENDIF
EQU	Equate Label to Variable Value
EXPORT	Export
IF	Conditional Assembly
INCLUDE	Include Source File
MACRO/ENDM	Macro Definition Start/End
ORG	Area Origin

6.1. Area - AREA

The AREA directive "name" is the name you give to this area. The linker gathers all areas with the same name together from different .c and .asm files. "memtype" is either ROM or RAM, and "mode" is either REL or ABS. RAM memtype cannot have initialized data (including code) and the assembler uses the memtype to warn illegal usage. Only an area with ABS mode can have the ORG directive. Areas with REL mode are concatenated by the linker.

```
label: AREAname(memtype,mode) ;comment
```

This directive **AREA <name> [(attributes)]** also defines a memory region to load the following code or data. The linker gathers all areas with the same name together and either concatenates or overlays them depending on the attributes. The attributes are:

abs, or <- absolute area
rel <- relocatable area

followed by

con, or <- concatenated
ovr <- overlay

The starting address of an absolute area is specified within the assembly file itself whereas the starting address of a relocatable area is specified as a command option to the linker. For an area with the “con” attribute, the linker concatenates areas of that name one after another. For an area with the “ovr” attribute, for each file, the linker starts an area at the same address. The following illustrates the differences:

```
file1.o:
    area text <- 10 bytes, call this text_1
    area data <- 10 bytes
    area text <- 20 bytes, call this text_2
file2.o:
    area data <- 20 bytes
    area text <- 40 bytes, call this text_3
```

text_1, text_2, and so on are just names used in this example. In practice, they are not given individual names. Let’s assume that the starting address of the text area is set to zero. Then, if the text area has the “con” attribute, text_1 would start at 0, text_2 at 10, and text_3 at 30. If the text area has the “ovr” attribute, then text_1 and text_2 would again have the addresses 0 and 10 respectively. text_3, since it starts in another file, would also have 0 as the starting address.

All areas of the same name must have the same attributes, even if they are used in different modules. Here are examples of the complete permutations of all acceptable attributes:

```
area foo(abs)
area foo(abs,con)
area foo(abs,ovr)
area foo(rel)
area foo(rel,con)
area foo(rel,ovr)
ascii “string”
ds “string”
dsu “string”
asciz “string”
```

6.2. RAM Block - BLK

The RAM Block directive reserves blocks of RAM in bytes.

```
label:   BLK<expr>   ;comment
```

The operand is an expression, specifying the size of the block (in bytes) to reserve. BLKB is synonymous with BLK, and BLKW is to reserve blocks in words. The AREA directive must be used to ensure the block of bytes will reside in the correct memory location. PSoC Designer requires that the 'bss' area be used for RAM variable. The following is an example of declaring the variable 'x':

```
AREA bss(ram)
x: BLK

AREA text
```

Note: Remember to change AREA to 'text' upon the conclusion of defining your RAM variables.

```
area bss(RAM)                ;inform assembler that variables follow
TX8_1_bytecount:   blk   1   ;declare local variable to index byte
                    ;to TX
countL:            blk   1   ;declare local variable to hold delay
                    ;value

area  text(ROM,REL)         ;inform assembler that relocatable
                             ;program code follows
```

6.3. RAM Block in Words - BLKW

The RAM Block in Words directive is to reserve blocks of RAM in words.

```
label:   BLKW<expr>   ;comment
```

```
area bss(RAM)                ;inform assembler that variables follow
countword:            blkw  1   ;declare local 16-bit variable to hold
                             ;delay value

area  text(ROM,REL)         ;inform assembler that relocatable
                             ;program code follows
```


6.4. Define Byte - DB

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.

```
label:  DB      operand1, operand2, ... operand(n) ;comment
```

The operands may be constant or a label. The number of operands in a DB statement can be zero, up to as many as will fit on the source line.

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

6.5. Define ASCII String - DS

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

```
label:  DS      "String of characters" ;comment
```

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 or use ASCIZ.

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

ASCII is synonymous with DS. ASCIZ can be used to define a NUL terminated string.

6.6. Define UNICODE String - DSU

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 "".

```
label: DSU "String of characters " ;comment
```

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 \. A quotation mark " can be entered into the middle of a string as \".

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.

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

6.7. Define Word - DW

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.

```
label: DW operand1, operand2, ... operand(n) ;comment
```

The operands may be constant or a label. The length of the source line limits the number of operands in a DW statement.

```
00D1 FF FE [00] tab2: DW -2
00D3 01 DF [00] DW 01DFh
00D5 00 11 [00] DW x
00D7 x: EQU 11h
```

6.8. Define Word, Little Endian Ordering - DWL

The Define Word directive reserves two bytes of ROM and assigns the specified words to the reserved two bytes, swapping the upper and lower bytes.

```
label: DWL operand1, operand2, ... operand(n) ;comment
```

The operands may be constant or a label. The length of the source line limits the number of operands in a DWL statement.

```
00D1 FE FF [00] tab3: DWL -2
00D3 DF 01 [00] DWL 01DFh
00D5 11 00 [00] DWL y
00D7 y: EQU 11h
```

6.9. Alternative Result of IF...ELSE...ENDIF - ELSE

The ELSE directive delineates a “Not True” action for a previous IF directive.

```
label: ELSE ;comment
```

```
--- 0000          DEBUG: EQU 0
--- 0014          _main:
                  IF (DEBUG)
                    mov a, 90h

--- 000ELSE
--- 0014 5000          mov a, 0
--- 000ENDIF
```

6.10. IF...ELSE...ENDIF - ENDIF

The ENDIF directive finishes a section of conditional assembly.

```
label: ENDIF value ;comment
```

```
--- 0000          DEBUG: EQU 0
--- 0014          _main:
                  IF (DEBUG)
                    mov a, 90h

--- 000ELSE
--- 0014 5000          mov a, 0
--- 000ENDIF
```

6.11. Equate Label - EQU

The Equate Label directive is used to assign an integer value to a label.

```
label: EQU operand ;comment
```

The label and operand are required for an EQU directive. The operand must be a constant or label or . (dot, the Program Counter). Each EQU directive may have only one operand and if a label is defined more than once, an assembly error will occur.

```
00D4 10 [00] DB zz
00D5 00 11 [00] DW YY ;Example of how label is used
00D7 xx: EQU 10h
00D7 YY: EQU 11h
00D7 zz: EQU xx
```

6.12. Export - EXPORT

The EXPORT directive is used to designate that a label can be referenced in another file. Otherwise, the label is not visible to another file. Another method to achieve this is to end a label definition with two colons instead of one.

```
EXPORT label ;comment
```

EXPORT foo and bar:: ret can also be referenced in separate files.

```
foo: MOV A,X
bar:: ret
```

6.13. IF...ELSE...ENDIF - IF

All source lines between the IF and ENDIF (or IF and ELSE) directives are assembled if the condition is true.

```
label: IF value ;comment
```

The following example shows a simple usage for IF, ELSE, and ENDIF. Since the label DEBUG' is set to zero the mov a, 90h did not generate code because the IF condition was not met (e.g., DEBUG was not > 0). The ELSE condition was then taken, and code was generated.

```
--- 0000 DEBUG: EQU 0
--- 0000 _main:
IF (DEBUG)
    mov a, 90h
--- 000ELSE
--- 0000 5000 mov a, 0h
--- 000ENDIF
```

6.14. Include Source File - INCLUDE

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

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.

```

//Port 0 (8-wide)
--- 0000      PRT0DR: EQU 00h      ;Port 0 data register      MEM BANK
--- 0001      PRT0IM: EQU 01h      ;Port 0 interrupt mask      0
--- 0002      PRT0BP: EQU 02h      ;Port 0 bypass enable      0
--- 0000      PRT0DM0: EQU 00h      ;Port 0 drive mode 0      1

// END

```

6.15. Macro Definition Start - MACRO/Macro Definition End – ENDM

The MACRO and ENDM directives are used to specify the start and end of a macro definition. The lines of code defined between a MACRO statement and an ENDM statement is not directly assembled into the program. Instead, it forms a macro that can later be substituted into the code by a macro call. Following the MACRO directive is the name used to call the macro as well as a list of parameters. Each time a parameter is used in the macro body of a macro call, it will be replaced by the corresponding value from the macro call.

Any assembly statement is allowed in a macro body except for another macro statement. Within a macro body, the expression @digit, where digit is between 0 and 9, is replaced by the corresponding macro argument when the macro is invoked.

You cannot define a macro name that conflicts with an instruction mnemonic or an assembly directive.

Following, is an example of an implicit macro:

Defines a macro named "foo:"

```
macro foo
mov @0,24
mov @1,@0
ENDM
```

Invoking foo with two arguments...

```
foo A,X
```

is equivalent to writing:

```
mov A,24
mov X,A
```

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

6.16. Area Origin - ORG

The ORG directive allows the programmer to set the value of the Program/Data Counter during assembly. This is most often used to set the start of a table in conjunction with the define directives DB, DS, and DW. The ORG directive can only be used in the area with the ABS mode.

label:	ORG	operand	;comment
--------	-----	---------	----------

The operand is required for an ORG directive and may be an integer constant, a label, or . (dot, the Program Counter). The Assembler does not keep track of areas previously defined and will not flag overlapping areas in a single source file.

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

This page has intentionally been left blank.

Section 7. Instruction Set

In this section you will learn (or can reference) the instruction set for the M8C.

All instructions are 1, 2, or 3 bytes wide and fetched from program memory, in a separate address space from data memory. The first byte of an instruction is an 8-bit constant, referred to as the Opcode. Depending on the instruction, there can be one or two succeeding bytes that encode address or operand information.

The following notation will be used throughout this section:

Notation	Description
A	Primary Accumulator
CF	Carry Flag
expr	Expression
F	Flags (ZF, CF, and Others)
I	Operand 1 Value
K	Operand 2 Value
PC	(PCH,PCL)
SP	Stack Pointer
X	X Register
ZF	Zero Flag

To access a complete instruction in detail within PSoC Designer, click your cursor on the target instruction in the file and hit **[F1]**.

7.1. Add with Carry

ADC

Add with Carry: **ADC**

Description: Adds the content of the Carry flag and destination with the source and places the result in the destination.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
ADC	A	expr	09h	Immediate		4
ADC	A	[expr]	0Ah	Direct Address		6
ADC	A	[X+expr]	0Bh	Index		7
ADC	[expr]	A	0Ch	Direct Address		7
ADC	[X+expr]	A	0Dh	Index		8
ADC	[expr]	expr	0Eh	Direct Address	Immediate	9
ADC	[X+expr]	expr	0Fh	Index	Immediate	10

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

 ZF Set if the result is zero; cleared otherwise.

Notes:

Example: or F,FlagCarry
 adc A,12

7.2. Add without Carry

ADD

Add without Carry: ADD

Description: Adds the content of the destination with the source and places the result in the destination.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
ADD	A	expr	01h	Immediate		4
ADD	A	[expr]	02h	Direct Address		6
ADD	A	[X+expr]	03h	Index		7
ADD	[expr]	A	04h	Direct Address		7
ADD	[X+expr]	A	05h	Index		8
ADD	[expr]	expr	06h	Direct Address	Immediate	9
ADD	[X+expr]	expr	07h	Index	Immediate	10
ADD	SP	expr	38h	Immediate		5

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

ZF Set if the result is zero; cleared otherwise. (ADD SP, expr does not affect the flags).

Notes:

Example: add A, 12

7.3. Bitwise AND

AND

Bitwise AND: **AND**

Description: A bitwise AND of the destination and source.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
AND	A	expr	21h	Immediate		4
AND	A	[expr]	22h	Direct Address		6
AND	A	[X+expr]	23h	Index		7
AND	[expr]	A	24h	Direct Address		7
AND	[X+expr]	A	25h	Index		8
AND	[expr]	expr	26h	Direct Address	Immediate	9
AND	[X+expr]	expr	27h	Index	Immediate	10
AND	REG[expr]	expr	41h	REG Direct Address	Immediate	9
AND	REG[X+expr]	expr	42h	REG Index	Immediate	10
AND	F	expr	70h	Immediate		4

Condition Flags: CF Unchanged (unless *F* is destination).

 ZF Set if the result is zero; cleared otherwise (unless *F* is destination).

Notes:

Example: and F, ~FlagCarry

7.4. Arithmetic Shift Left

ASL

Arithmetic Shift Left: ASL

Description: Shifts all bits of specified location one place to the left. The most significant is loaded into the Carry flag. Bit 0 is loaded with a zero.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
ASL	A		64h			4
ASL	[expr]		65h	Direct Address		7
ASL	[X+expr]		66h	Index		8

Condition Flags: CF Set if the MSB of the source was set before the shift, cleared otherwise.

ZF Set if the result is zero; cleared otherwise.

Notes:

Example: `asl A ; Multiply by 2 and put MSB in Carry flag`

7.5. Arithmetic Shift Right

ASR

Arithmetic Shift Right: ASR

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

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
ASR	A		67h			4
ASR	[expr]		68h	Direct Address		7
ASR	[X+expr]		69h	Index		8

Condition Flags: CF Set if LSB of the source was set before the shift, cleared otherwise.

ZF Set if the result is zero; cleared otherwise.

Notes:

Example: `asr A ; Divide by two`

7.6. Call Function

CALL

Call Function: CALL

Description: Executes a jump to a subroutine starting at the address given as an operand. The Program Counter (PC) is pushed onto the stack. The stack pointer is post-incremented. The PC is loaded with the address value. This instruction has a 12-bit-two's-complement-relative-address that is added to the PC to form the jump target. In contrast, the Long Call instruction has two operands that together form a 16-bit absolute address.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
CALL	Address		90h	Address		11
CALL	Address		91h	Address		11
CALL	Address		92h	Address		11
CALL	Address		93h	Address		11
CALL	Address		94h	Address		11
CALL	Address		95h	Address		11
CALL	Address		96h	Address		11
CALL	Address		97h	Address		11
CALL	Address		98h	Address		11
CALL	Address		99h	Address		11
CALL	Address		9Ah	Address		11
CALL	Address		9Bh	Address		11
CALL	Address		9Ch	Address		11
CALL	Address		9Dh	Address		11
CALL	Address		9Eh	Address		11
CALL	Address		9Fh	Address		11

Condition Flags: CF Carry flag unaffected.

ZF Zero flag unaffected.

Notes: This is a 12-bit address. The upper 4 bits of the address come from the lower (nibble) 4 bits of the Opcode. (9xh where x = lower 4 bits.) The lower 8 bits of the address come from Operand 1.

Example:

```
call LoadConfig
.
.
LoadConfig:
ret
```

7.7. Non-destructive Compare

CMP

Non-destructive Compare: **CMP**

Description: Subtracts two operands and records the result in flags. The contents of the accumulator are unaffected.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
CMP	A	expr	39h	Immediate		5
CMP	A	[expr]	3Ah	Direct Address		7
CMP	A	[X+expr]	3Bh	Index		8
CMP	[expr]	expr	3Ch	Direct Address	Immediate	8
CMP	[X+expr]	expr	3Dh	Index	Immediate	9

Condition Flags: CF Set if the Operand 1 < Operand 2 value; cleared otherwise.

ZF Set if the operands are equal; cleared otherwise.

Notes:

Example: `cmp A, END_CONFIG_TABLE`

7.8. Complement Accumulator

CPL

Complement Accumulator: **CPL**

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

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
CPL	A		73h			4

Condition Flags: CF Unchanged.

ZF Set if the result is zero; cleared otherwise.

Notes:

Example: `cpl A`

7.9. Decrement

DEC

Decrement: DEC

Description: Subtract one from the contents of a register or Data RAM space. The field to the right of the Opcode determines which entity is affected: accumulator; x register; direct or index addressed Data RAM location.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
DEC	A		78h			4
DEC	X		79h			4
DEC	[expr]		7Ah	Direct Address		7
DEC	[X+expr]		7Bh	Index		8

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

ZF Set if the result is zero; cleared otherwise.

Notes:

Example:

```

loop2:                                ;loop takes 12 CPU cycles
      dec  [countL]
      jnz  loop2

```

7.10. Halt

HALT

Halt: HALT

Description: Halts the execution of the processor. The processor will remain halted until a Power On Reset (POR) or Watchdog Timer Reset (WDR) event happens, either of which will cause the processor to begin execution again.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
HALT			30h			

Condition Flags: CF Carry flag unaffected.

ZF Zero flag unaffected.

Notes:

Example: halt

7.11. Increment

INC

Increment: INC

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 affected: accumulator; X register; direct or index addressed Data RAM location.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
INC	A		74h			4
INC	X		75h			4
INC	[expr]		76h	Direct Address		7
INC	[X+expr]		77h	Index		8

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

ZF Set if the result is zero; cleared otherwise.

Notes:

Example: `inc X`

7.12. Table Read

INDEX

Table Read: INDEX

Description: Places the contents of ROM location indexed by the sum of the accumulator and the address operand, into the accumulator. This instruction has a 12-bit-twos-complement-address, relative to the PC as it points to the next instruction.

This instruction is used to move information in table form from the program memory space into the accumulator. It has one operand, which is the lower part of the base address of a program memory table. The lower nibble of the INDEX Opcode forms the upper part of the base address. This 12-bit value is a sign - extended 12-bit-twos-complement-address that when added to the PC + 2, yields the base address of the table.

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.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
INDEX	Address		F0h	Address Offset		13
INDEX	Address		F1h	Address Offset		13
INDEX	Address		F2h	Address Offset		13
INDEX	Address		F3h	Address Offset		13
INDEX	Address		F4h	Address Offset		13
INDEX	Address		F5h	Address Offset		13
INDEX	Address		F6h	Address Offset		13
INDEX	Address		F7h	Address Offset		13
INDEX	Address		F8h	Address Offset		13
INDEX	Address		F9h	Address Offset		13
INDEX	Address		FAh	Address Offset		13
INDEX	Address		FBh	Address Offset		13
INDEX	Address		FCh	Address Offset		13
INDEX	Address		FDh	Address Offset		13
INDEX	Address		FEh	Address Offset		13
INDEX	Address		FFh	Address Offset		13

Condition Flags: CF Unchanged.

ZF Set if A is zero.

Notes: This is a 12-bit address. The upper 4 bits of the address come from the lower (nibble) 4 bits of the Opcode. (F_xh where x = lower 4 bits.) The lower 8 bits of the address come from Operand 1.

Example:

```
inc    A                ;increment index value for next TX
index TXoutput         ;retrieve indexed TX value
ret
TXoutput:
    db 00h,55h,ffh,55h ;data sent to TX
```

7.13. Jump Accumulator

JACC

Jump Accumulator: JACC

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. This instruction has a 12-bit-two's-complement-relative-address that is added to the PC to form the jump target. In contrast, the Long Jump instruction has two operands that together form a 16-bit absolute address.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
JACC	Address		E0h	Address		7
JACC	Address		E1h	Address		7
JACC	Address		E2h	Address		7
JACC	Address		E3h	Address		7
JACC	Address		E4h	Address		7
JACC	Address		E5h	Address		7
JACC	Address		E6h	Address		7
JACC	Address		E7h	Address		7
JACC	Address		E8h	Address		7
JACC	Address		E9h	Address		7
JACC	Address		EAh	Address		7
JACC	Address		EBh	Address		7
JACC	Address		ECh	Address		7
JACC	Address		EDh	Address		7
JACC	Address		EEh	Address		7
JACC	Address		EFh	Address		7

Condition Flags: CF Unchanged.

ZF Unchanged.

Notes: This is a 12-bit address. The upper 4 bits of the address come from the lower (nibble) 4 bits of the Opcode. (E x h where x = lower 4 bits.) The lower 8 bits of the address come from Operand 1.

Example: `jacc maintable`

7.14. Jump if Carry

JC

Jump if Carry: JC

Description: If the carry flag is set (1 = true), jump to the address (place the address in the Program Counter). This instruction has a 12-bit-two's-complement-relative-address that is added to the PC to form the jump target. In contrast, the Long Jump instruction has two operands that together form a 16-bit absolute address.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
JC	Address		C0h	Address		4/5*
JC	Address		C1h	Address		4/5
JC	Address		C2h	Address		4/5
JC	Address		C3h	Address		4/5
JC	Address		C4h	Address		4/5
JC	Address		C5h	Address		4/5
JC	Address		C6h	Address		4/5
JC	Address		C7h	Address		4/5
JC	Address		C8h	Address		4/5
JC	Address		C9h	Address		4/5
JC	Address		CAh	Address		4/5
JC	Address		CBh	Address		4/5
JC	Address		CCh	Address		4/5
JC	Address		CDh	Address		4/5
JC	Address		CEh	Address		4/5
JC	Address		CFh	Address		4/5

Condition Flags: CF Carry flag unaffected.

ZF Zero flag unaffected.

Notes: This is a 12-bit address. The upper 4 bits of the address come from the lower (nibble) 4 bits of the Opcode. (Cxh where x = lower 4 bits.) The lower 8 bits of the address come from Operand 1.

** If the carry flag is true (1), 5 cycles are consumed because the jump is taken. If the carry flag is false (0), 4 cycles are consumed because the jump is not taken.*

Example:

```

cmp    [TX8_1_bytecount],4    ;rotate through 4 TX values
jc     branch2

```

7.15. Jump

JMP

Jump: **JMP**

Description: Jump unconditionally to the address (place the address in the Program Counter). This instruction has a 12-bit-two's-complement-relative-address that is added to the PC to form the jump target. In contrast, the Long Jump instruction has two operands that together form a 16-bit absolute address.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
JMP	Address		80h	Address		5
JMP	Address		81h	Address		5
JMP	Address		82h	Address		5
JMP	Address		83h	Address		5
JMP	Address		84h	Address		5
JMP	Address		85h	Address		5
JMP	Address		86h	Address		5
JMP	Address		87h	Address		5
JMP	Address		88h	Address		5
JMP	Address		89h	Address		5
JMP	Address		8Ah	Address		5
JMP	Address		8Bh	Address		5
JMP	Address		8Ch	Address		5
JMP	Address		8Dh	Address		5
JMP	Address		8Eh	Address		5
JMP	Address		8Fh	Address		5

Condition Flags: CF Carry flag unaffected.

 ZF Zero flag unaffected.

Notes: This is a 12-bit address. The upper 4 bits of the address come from the lower (nibble) 4 bits of the Opcode. (8xh where x = lower 4 bits.) The lower 8 bits of the address come from Operand 1.

Example: loop:
 cmp [__r0],<__bss_end
 jz done
 jmp loop
 done:

7.16. Jump if No Carry

JNC

Jump if No Carry: JNC

Description: If the carry flag is **not** set (0 = false), jump to the address (place the address in the Program Counter). This instruction has a 12-bit-two's-complement-relative-address that is added to the PC to form the jump target. In contrast, the Long Jump instruction has two operands that together form a 16-bit absolute address.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
JNC	Address		D0h	Address		4/5*
JNC	Address		D1h	Address		4/5
JNC	Address		D2h	Address		4/5
JNC	Address		D3h	Address		4/5
JNC	Address		D4h	Address		4/5
JNC	Address		D5h	Address		4/5
JNC	Address		D6h	Address		4/5
JNC	Address		D7h	Address		4/5
JNC	Address		D8h	Address		4/5
JNC	Address		D9h	Address		4/5
JNC	Address		DAh	Address		4/5
JNC	Address		DBh	Address		4/5
JNC	Address		DCh	Address		4/5
JNC	Address		DDh	Address		4/5
JNC	Address		DEh	Address		4/5
JNC	Address		DFh	Address		4/5

Condition Flags: CF Carry flag unaffected.

ZF Zero flag unaffected.

Notes: This is a 12-bit address. The upper 4 bits of the address come from the lower (nibble) 4 bits of the Opcode. (Dxh where x = lower 4 bits.) The lower 8 bits of the address come from Operand 1.

** If the carry flag is false (0), 5 cycles are consumed because the jump is taken. If the carry flag is true (1), 4 cycles are consumed because the jump is not taken.*

Example:

```

cmp    [TX8_1_bytecount],4    ;rotate through 4 TX values
jnc    branch2

```

7.17. Jump if Not Zero

JNZ

Jump if Not Zero: JNZ

Description: If the zero flag is **not** set (0 = false), jump to the address. This instruction has a 12-bit-two's-complement-relative-address that is added to the PC to form the jump target. In contrast, the Long Jump instruction has two operands that together form a 16-bit absolute address.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
JNZ	Address		B0h	Address		4/5*
JNZ	Address		B1h	Address		4/5
JNZ	Address		B2h	Address		4/5
JNZ	Address		B3h	Address		4/5
JNZ	Address		B4h	Address		4/5
JNZ	Address		B5h	Address		4/5
JNZ	Address		B6h	Address		4/5
JNZ	Address		B7h	Address		4/5
JNZ	Address		B8h	Address		4/5
JNZ	Address		B9h	Address		4/5
JNZ	Address		BAh	Address		4/5
JNZ	Address		BBh	Address		4/5
JNZ	Address		BCh	Address		4/5
JNZ	Address		BDh	Address		4/5
JNZ	Address		BEh	Address		4/5
JNZ	Address		BFh	Address		4/5

Condition Flags: CF Carry flag unaffected.

ZF Zero flag unaffected.

Notes: This is a 12-bit address. The upper 4 bits of the address come from the lower (nibble) 4 bits of the Opcode. (Bxh where x = lower 4 bits.) The lower 8 bits of the address come from Operand 1.

** If the zero flag is false (0), 5 cycles are consumed because the jump is taken. If the zero flag is true (1), 4 cycles are consumed because the jump is not taken.*

Example:

```
mov    a, [_c32c]
jnz    lp1
```

7.18. Jump if Zero

JZ

Jump if Zero: JZ

Description: If the zero flag is set (1 = true), jump to the address (add the address to the Program Counter). This instruction has a 12-bit-two's-complement-relative-address that is added to the PC to form the jump target. In contrast, the Long Jump instruction has two operands that together form a 16-bit absolute address.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
JZ	Address		A0h	Address		4/5*
JZ	Address		A1h	Address		4/5
JZ	Address		A2h	Address		4/5
JZ	Address		A3h	Address		4/5
JZ	Address		A4h	Address		4/5
JZ	Address		A5h	Address		4/5
JZ	Address		A6h	Address		4/5
JZ	Address		A7h	Address		4/5
JZ	Address		A8h	Address		4/5
JZ	Address		A9h	Address		4/5
JZ	Address		AAh	Address		4/5
JZ	Address		ABh	Address		4/5
JZ	Address		ACH	Address		4/5
JZ	Address		ADh	Address		4/5
JZ	Address		Aeh	Address		4/5
JZ	Address		Afh	Address		4/5

Condition Flags: CF Carry flag unaffected.

ZF Zero flag unaffected.

Notes: This is a 12-bit address. The upper 4 bits of the address come from the lower (nibble) 4 bits of the Opcode. (Axh where x = lower 4 bits.) The lower 8 bits of the address come from Operand 1.

*If the zero flag is true (1), 5 cycles are consumed because the jump is taken.
If the zero flag is false (0), 4 cycles are consumed because the jump is not taken.*

Example:

```

cmp    A, END_CONFIG_TABLE    ;check for end of table
jz     EndLoadConfig          ;if so, end of load

```


7.19. Long Call

LCALL

Long Call: LCALL

Description: Executes a jump to a subroutine starting at the address given as an operand. The PC is pushed onto the stack. The stack pointer is post-incremented. The PC is loaded with the address value. The Long Call instruction has two operands that together form a 16-bit absolute address. The other call instruction has a 12-bit-two-complement-relative-address that is added to the PC to form the jump target.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
LCALL	Address		7Ch	16-bit Address (High Byte)	16-bit Address (Low Byte)	13

Condition Flags: CF

ZF

Notes: Again, LCALL is a 16-bit address. Its instruction is identical to a CALL but can jump to a subroutine over a wider address range.

Example: `lcall PWMDB8_1INT`

7.20. Long Jump

LJMP

Long Jump: LJMP

Description: Jump unconditionally to the address (place the address in the Program Counter). The Long Jump instruction has two operands that together form a 16-bit absolute address. All other jump instructions have 12-bit-two-complement-relative-addresses that are added to the PC to form the jump target.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
LJMP	Address		7Dh	16-bit Address (High Byte)	16-bit Address (Low Byte)	7

Condition Flags: CF

ZF

Notes: Again, LJMP is a 16-bit address. Its instruction is identical to a JMP but can jump over a wider address range.

Example: `ljmp PWMDB8_1INT`

7.21. Move

MOV

Move: MOV

Description: This instruction allows for a number of combinations of moves. Immediate, direct, and indexed addressing are supported.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/ Byte 1	Byte 2	Byte 3	
MOV	X	SP	4Fh			4
MOV	A	expr	50h	Immediate		4
MOV	A	[expr]	51h	Direct Address		5
MOV	A	[X+expr]	52h	Index		6
MOV	[expr]	A	53h	Direct Address		5
MOV	[X+expr]	A	54h	Index		6
MOV	[expr]	expr	55h	Direct Address	Immediate	8
MOV	[X+expr]	expr	56h	Index	Immediate	9
MOV	X	expr	57h	Immediate		4
MOV	X	[expr]	58h	Direct Address		6
MOV	X	[X+expr]	59h	Index		7
MOV	[expr]	X	5Ah	Direct Address		5
MOV	A	X	5Bh			4
MOV	X	A	5Ch			4
MOV	A	REG[expr]	5Dh	REG Direct Address		6
MOV	A	REG[X+expr]	5Eh	REG Index		7
MOV	[expr]	[expr]	5Fh	Direct Address	Direct Address	10
MOV	REG[expr]	A	60h	REG Direct Address		5
MOV	REG[X+expr]	A	61h	REG Index		6
MOV	REG[expr]	expr	62h	REG Direct Address	Immediate	8
MOV	REG[X+expr]	expr	63h	REG Index	Immediate	9

Condition Flags: CF Carry flag unaffected.

ZF Set if A is updated with zero.

Notes:

Example:

```

mov  A, ProjectName
mov  X, ProjectName

```

7.22. Move Indirect, Post-Increment to Memory

MVI

Move Indirect, Post-Increment to Memory: MVI

Description:

The address for memory is specified by the contents of memory at the direct address given by the operand. Memory is indirectly read or written to by this address. The contents of the original direct memory location is incremented and can only be located in the first SRAM page. The indirect address is wherever the Extended Addressing points are located.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/ Byte 1	Byte 2	Byte 3	
MVI	A	[expr]	3Eh	Direct Address (Page 0)		10
MVI	[expr]	A	3Fh	Direct Address (Page 0)		10

Condition Flags: CF Unaffected.

ZF Set if A is updated with zero.

Notes:

Example: `mvi [location],A`

7.23. No Operation

NOP

No Operation: NOP

Description: This one-byte instruction performs no operation.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
NOP			40h			4

Condition Flags: CF Carry flag unaffected.

ZF Zero flag unaffected.

Notes:

Example: nop

7.24. Bitwise OR

OR

Bitwise OR: OR

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.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
OR	A	expr	29h	Immediate		4
OR	A	[expr]	2Ah	Direct Address		6
OR	A	[X+expr]	2Bh	Index		7
OR	[expr]	A	2Ch	Direct Address		7
OR	[X+expr]	A	2Dh	Index		8
OR	[expr]	expr	2Eh	Direct Address	Immediate	9
OR	[X+expr]	expr	2Fh	Index	Immediate	10
OR	REG[expr]	expr	43h	REG Direct Address	Immediate	9
OR	REG[X+expr]	expr	44h	REG Index	Immediate	10
OR	F	expr	71h	Immediate		4

Condition Flags: CF Unchanged (unless *F* is destination).

ZF Set if the result is zero; cleared otherwise (unless *F* is destination).

Notes:

Example: or F, FLAG_CFG_MASK

7.25. Pop Stack into Register

POP

Pop Stack into Register: POP

Description: Place the contents of the top of the stack into the designated register(s). Pre-decrement the stack pointer.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
POP	A		18h			5
POP	X		20h			5

Condition Flags: CF Carry flag unaffected.
ZF Set if A is updated to zero.

Notes:

Example: pop A
 reti

7.26. Push Register onto Stack

PUSH

Push Register onto Stack: PUSH

Description: Push the contents of the designated register(s) onto the stack. Post-increment the stack pointer.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
PUSH	A		08h			4
PUSH	X		10h			4

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

Notes:

Example: push A

7.27. Return

RET

Return: RET

Description: Pop two bytes off of the stack into the Program Counter.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
RET			7Fh			8

Condition Flags: CF Carry unchanged by this instruction.

ZF Zero unchanged by this instruction.

Notes:

Example: ret

7.28. Return from Interrupt

RETI

Return from Interrupt: RETI

Description: Pop flag then pop two bytes off of the stack into the Program Counter. The previous flag interrupt enable status is restored. Do not use an RETI to return from a subroutine call.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
RETI			7Eh			10

Condition Flags: CF All flag bits are restored to the value pushed during an interrupt call.

ZF All flag bits are restored to the value pushed during an interrupt call.

Notes: Flags restored.

Example: Interrupt6:
reti

7.29. Rotate Left through Carry

RLC

Rotate Left through Carry: RLC

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

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
RLC	A		6Ah			4
RLC	[expr]		6Bh	Direct Address		7
RLC	[X+expr]		6Ch	Index		8

Condition Flags: CF Set if the MSB of the specified accumulator was set before the shift.

ZF Set if the result is zero; cleared otherwise.

Notes:

Example: `rlc [shiftout]`

7.30. Table Read

ROMX

Table Read: ROMX

Description: Places the contents of ROM into the accumulator, indexed by the address generated by concatenating the A and X registers (A is Most Significant and X is Least Significant).

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
ROMX			28h			11

Condition Flags: CF Unchanged.

ZF Set if A is zero.

Notes:

Example:

```

mov    A, 42
inc    X
romx           ;load config address

```


7.31. Rotate Right through Carry

RRC

Rotate Right through Carry: RRC

Description:

Shifts all bits of the specified location one place to the right. The carry flag is loaded into the most significant bit of the specified location - bit 7. Bit 0 of the source is loaded into the carry flag.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
RRC	A		6Dh			4
RRC	[expr]		6Eh	Direct Address		7
RRC	[X+expr]		6Fh	Index		8

Condition Flags: CF Set if LSB of the specified accumulator was set before the shift.

ZF Set if the result is zero; cleared otherwise.

Notes:

Example: `rrc [shiftout]`

7.32. Subtract with Borrow

SBB

Subtract with Borrow: SBB

Description: Subtracts a value; K, [K] or [X + K], plus the carry flag, from the destination contents and places the result in the destination.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
SBB	A	expr	19h	Immediate		4
SBB	A	[expr]	1Ah	Direct Address		6
SBB	A	[X+expr]	1Bh	Index		7
SBB	[expr]	A	1Ch	Direct Address		7
SBB	[X+expr]	A	1Dh	Index		8
SBB	[expr]	expr	1Eh	Direct Address	Immediate	9
SBB	[X+expr]	expr	1Fh	Index	Immediate	10

Condition Flags: CF Set if, treating the numbers as unsigned, the result < 0; cleared otherwise.

ZF Set if the result is zero; cleared otherwise.

Notes:

Example: `sbb A, [oldvalue]`

7.33. Subtract without Borrow

SUB

Subtract without Borrow: SUB

Description: Subtracts a value; K, [K] or [X + K] from the contents of the destination.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
SUB	A	expr	11h	Immediate		4
SUB	A	[expr]	12h	Direct Address		6
SUB	A	[X+expr]	13h	Index		7
SUB	[expr]	A	14h	Direct Address		7
SUB	[X+expr]	A	15h	Index		8
SUB	[expr]	expr	16h	Direct Address	Immediate	9
SUB	[X+expr]	expr	17h	Index	Immediate	10

Condition Flags: CF Set if, treating the numbers as unsigned, the result < 0; cleared otherwise.

ZF Set if the result is zero; cleared otherwise.

Notes:

Example: `sub A, 31 ; Apply the offset`

7.34. Swap

SWAP

Swap: SWAP

Description: Operates on X register, memory, or the stack pointer with the primary accumulator. An internal temporary register is used to facilitate the swap.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
SWAP	A	X	4Bh			5
SWAP	A	[expr]	4Ch	Direct Address		7
SWAP	X	[expr]	4Dh	Direct Address		7
SWAP	A	SP	4Eh			5

Condition Flags: CF Carry flag unaffected.

ZF Set if accumulator is cleared.

Notes:

Example: swap SP,A

7.35. System Supervisor Call

SSC

System Supervisor Call: SSC

Description: The System Supervisor Call instruction provides the method for the user to access pre-existing routines in the Supervisor ROM to invoke various system-related functions. The user must set several parameters when utilizing these functions. The parameters are written to a block near the top of RAM memory space.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
SSC			00h			4

Condition Flags: CF Unchanged.

ZF Unchanged.

Notes: The functions and parameters are device-specific. See the device Data Sheet for details. (Access it at <http://www.cypressmicro.com>.)

Example:

7.36. Test with Mask

TST

Test with Mask: TST

Description: A Read-Only operation on an IO port or memory addressed by the first argument, with a bit mask given by the second argument. The operand is not modified.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
TST	[expr]	expr	47h	Direct Address	Immediate	8
TST	[X+expr]	expr	48h	Index	Immediate	9
TST	REG[expr]	expr	49h	REG Direct Address	Immediate	8
TST	REG[X+expr]	expr	4Ah	REG Index	Immediate	9

Condition Flags: CF Unaffected.

ZF Set if the addressed location is zero; cleared otherwise.

Notes:

Example: `tst reg[TX8_1_CONTROL_0],TX_REG_EMPTY`

7.37. Bitwise XOR

XOR

Bitwise XOR: XOR

Description: A Bitwise Exclusive OR of a value; K, [K] or [X + K] and the destination contents.

Source Format			Machine Code			Cycles
Instruction	Operand 1	Operand 2	Opcode/Byte 1	Byte 2	Byte 3	
XOR	A	expr	31h	Immediate		4
XOR	A	[expr]	32h	Direct Address		6
XOR	A	[X+expr]	33h	Index		7
XOR	[expr]	A	34h	Direct Address		7
XOR	[X+expr]	A	35h	Index		8
XOR	[expr]	expr	36h	Direct Address	Immediate	9
XOR	[X+expr]	expr	37h	Index	Immediate	10
XOR	REG[expr]	expr	45h	REG Direct Address	Immediate	9
XOR	REG[X+expr]	expr	46h	REG Index	Immediate	10
XOR	F	expr	72h	Immediate		4

Condition Flags: CF Unchanged (unless *F* is destination).

ZF Set if the result is zero; cleared otherwise (unless *F* is destination).

Notes:

Example: xor A, [bitmask]

Section 8. Compile/Assemble Error Messages

In this section you will learn (or can reference) all PSoC Designer C Compiler and Assembler errors and warnings.

Again, once you have added and modified assembly-language source and/or C Compiler files, you must compile/assemble the files and build the project. This is done so PSoC Designer can generate a .rom file to be used to debug the M8C program.



To compile the source files for the current project, click the **Compile/Assemble** icon in the toolbar.



To build the current project, click the **Build** icon in the toolbar.

Each time you compile/assemble files or build the project, the status window is cleared and the current status entered as the process occurs.

When compiling or building is complete, you will see the number of errors. Zero errors signify that the compilation/assembly or build was successful. One or more errors indicate problems with one or more files. For further information on the PSoC Designer status window refer to section 3 in the *PSoC Designer: Integrated Development Environment User Guide*.

The remainder of this section lists all compile/assemble and build (linker) errors and warnings you might encounter from your code.

8.1. Preprocessor

Note that these errors and warnings are associated with C Compiler errors and warnings.

Error/Warning
not followed by macro parameter
occurs at border of replacement
#defined token can't be redefined
#defined token is not a name
#elif after #else
#elif with no #if
#else after #else
#else with no #if
#endif with no #if
#if too deeply nested
#line specifies number out of range
Bad ?: in #if/endif
Bad syntax for control line
Bad token r produced by ## operator
Character constant taken as not signed
Could not find include file

(Preprocessor cont.)

Disagreement in number of macro arguments
Duplicate macro argument
EOF in macro arglist
EOF in string or char constant
EOF inside comment
Empty character constant
Illegal operator * or & in #if/#elsif
Incorrect syntax for 'defined'
Macro redefinition
Multibyte character constant undefined
Sorry, too many macro arguments
String in #if/#elsif
Stringified macro arg is too long
Syntax error in #else
Syntax error in #endif
Syntax error in #if/#elsif
Syntax error in #if/#endif
Syntax error in #ifdef/#ifndef
Syntax error in #include
Syntax error in #line
Syntax error in #undef
Syntax error in macro parameters
Undefined expression value
Unknown preprocessor control line
Unterminated #if/#ifdef/#ifndef
Unterminated string or char const

8.1.1. Preprocessor Command Line Errors

Error/Warning
Can't open input file
Can't open output file
Illegal -D or -U argument
Too many -I directives

8.2. C Compiler

Error/Warning
expecting <character>
literal too long
IO port <name> cannot be redeclared as local variable
IO port <name> cannot be redeclared as parameter
IO port variable <name> cannot have initializer
<n> is a preprocessing number but an invalid %s constant
<n> is an illegal array size
<n> is an illegal bit-field size
<type> is an illegal bit-field type
<type> is an illegal field type
'sizeof' applied to a bit field
addressable object required

(C Compiler cont.)

asm string too long
assignment to const identifier
assignment to const location
cannot initialize undefined
case label must be a constant integer expression
cast from <type> to <type> is illegal in constant expressions
cast from <type> to <type> is illegal
conflicting argument declarations for function <name>
declared parameter <name> is missing
duplicate case label <n>
duplicate declaration for <name> previously declared at <line>
duplicate field name <name> in <structure>
empty declaration
expecting an enumerator identifier
expecting an identifier
extra default label
extraneous identifier <id>
extraneous old-style parameter list
extraneous return value
field name expected
field name missing
found <id> expected a function
ill-formed hexadecimal escape sequence
illegal break statement
illegal case label
illegal character <c>
illegal continue statement
illegal default label
illegal expression
illegal formal parameter types
illegal initialization for <id>
illegal initialization for parameter <id>
illegal initialization of 'extern <name>'
illegal return type <type>
illegal statement termination
illegal type <type> in switch expression
illegal type 'array of <name>'
illegal use of incomplete type
illegal use of type name <name>
initializer must be constant
insufficient number of arguments to <function>
integer expression must be constant
interrupt handler <name> cannot have arguments
invalid field declarations
invalid floating constant
invalid hexadecimal constant
invalid initialization type; found <type> expected <type>
invalid octal constant
invalid operand of unary &; <id> is declared register
invalid storage class <storage class> for <id>

(C Compiler cont.)

invalid type argument <type> to 'sizeof'
invalid type specification
invalid use of 'typedef'
left operand of -> has incompatible type
left operand of . has incompatible type
lvalue required
missing <c>
missing tag
missing array size
missing identifier
missing label in goto
missing name for parameter to function <name>
missing parameter type
missing string constant in asm
missing { in initialization of <name>
operand of unary <operator> has illegal type
operands of <operator> have illegal types <type> and <type>
overflow in value for enumeration constant
redeclaration of <name> previously declared at <line>
redeclaration of <name>
redefinition of <name> previously defined at <line>
redefinition of label <name> previously defined at <line>
size of <type> exceeds <n> bytes
size of 'array of <type>' exceeds <n> bytes
syntax error; found
too many arguments to <function>
too many errors
too many initializers
too many variable references in asm string
type error in argument <name> to <function>; <type> is illegal
type error in argument <name> to <function>; found <type> expected <type>
type error
unclosed comment
undeclared identifier <name>
undefined label
undefined size for <name>
undefined size for field <name>
undefined size for parameter <name>
undefined static <name>
unknown #pragma
unknown size for type <type>
unrecognized declaration
unrecognized statement

8.3. Assembler

Error/Warning
'[' addressing mode must end with ']'
) expected
.if/.else/.endif mismatched
<character> expected
EOF encountered before end of macro definition
No preceding global symbol
absolute expression expected
badly formed argument, (without a matching)
branch out of range
cannot add two relocatable items
cannot perform subtract relocation
cannot subtract two relocatable items
cannot use .org in relocatable area
character expected
comma expected
equ statement must have a label
identifier expected, but got character <c>
illegal addressing mode
illegal operand
input expected
label must start with an alphabet, '.' or '_'
letter expected but got <c>
macro <name> already entered
macro definition cannot be nested
maximum <#> macro arguments exceeded
missing macro argument number
multiple definitions <name>
no such mnemonic <name>
relocation error
target too far for instruction
too many include files
too many nested .if
undefined mnemonic <word>
undefined symbol
unknown operator
unmatched .else
unmatched .endif

8.3.1. Assembler Command Line Errors

Error/Warning
cannot create output file %s\n
Too many include paths

8.4. Linker

Error/Warning
Address <address> already contains a value
can't find address for symbol <symbol>
can't open file <file>
can't open temporary file <file>
cannot open library file <file>
cannot write to <file>
definition of builtin symbol <symbol> ignored
ill-formed line <%s> in the listing file
multiple define <name>
no space left in section <area>
redefinition of symbol <symbol>
undefined symbol <name>
unknown output format <format>

M8C Instruction Set Reference Table

Program Flow Instructions				Flags	Cycles
CALL		Call (relative)			
9h	CALL addr	LSB Address Byte (MSN in opcode, x)			11
HALT		Halt			
30h	HALT				NA
JACC		Jump Accumulator (relative)			
Exh	JACC addr	LSB Address Byte (MSN in opcode, x)			7
JC		Jump if Carry (relative)			
Cxh	JC addr	LSB Address Byte (MSN in opcode, x)			5/4
JMP		Jump (relative)			
8xh	JMP addr	LSB Address Byte (MSN in opcode, x)			5
JNC		Jump if No Carry (relative)			
Dxh	JNC addr	LSB Address Byte (MSN in opcode, x)			5/4
JNZ		Jump if Not Zero (relative)			
Bxh	JNZ addr	LSB Address Byte (MSN in opcode, x)			5/4
JZ		Jump if Zero (relative)			
Axh	JZ addr	LSB Address Byte (MSN in opcode, x)			5/4
LCALL		Long Call			
7Ch	LCALL addr	addr MSB	addr LSB		13
LJMP		Long Jump			
7Dh	LJMP addr	addr MSB	addr LSB		7
NOP		No Operation			
40h	NOP				4
RET		Return from Call			
7Fh	RET				8
RETI		Return from Interrupt			
7Eh	RETI				10
SSC		System Supervisor Call			
00h	SSC				NA

Non Destructive Test Instructions				Flags	Cycles
CMP		Non Destructive Compare			
39h	CMP A,expr	Immediate		c z	5
3Ah	CMP A,[expr]	Direct Address		c z	7
3Bh	CMP A,[X+expr]	Index		c z	8
3Ch	CMP [expr],expr	Direct Address	Immediate	c z	8
3Dh	CMP [X+expr],expr	Index	Immediate	c z	9
TST		Test with Mask			
47h	TST [expr],expr	Direct Address	Bit Mask	z	8
48h	TST [X+expr],expr	Index	Bit Mask	z	9
49h	TST REG[expr],expr	Reg Direct Address	Bit Mask	z	8
4Ah	TST REG[X+expr],expr	Reg Index	Bit Mask	z	9

Arithmetic Instructions				Flags	Cycles
ADC		Add with Carry			
09h	ADC A,expr	Immediate		c z	4
0Ah	ADC A,[expr]	Direct Address		c z	6
0Bh	ADC A,[X+expr]	Index		c z	7
0Ch	ADC [expr],A	Direct Address		c z	7
0Dh	ADC [X+expr],A	Index		c z	8
0Eh	ADC [expr],expr	Direct Address	Immediate	c z	9
0Fh	ADC [X+expr],expr	Index	Immediate	c z	10
ADD		Add without Carry			
01h	ADD A,expr	Immediate		c z	4
02h	ADD A,[expr]	Direct Address		c z	6
03h	ADD A,[X+expr]	Index		c z	7
04h	ADD [expr],A	Direct Address		c z	7
05h	ADD [X+expr],A	Index		c z	8
06h	ADD [expr],expr	Direct Address	Immediate	c z	9
07h	ADD [X+expr],expr	Index	Immediate	c z	10
38h	ADD SP,expr	Immediate		c z	5
SBB		Subtract with Borrow			
19h	SBB A,expr	Immediate		c z	4
1Ah	SBB A,[expr]	Direct Address		c z	6
1Bh	SBB A,[X+expr]	Index		c z	7
1Ch	SBB [expr],A	Direct Address		c z	7
1Dh	SBB [X+expr],A	Index		c z	8
1Eh	SBB [expr],expr	Direct Address	Immediate	c z	9
1Fh	SBB [X+expr],expr	Index	Immediate	c z	10
SUB		Subtract without Borrow			
11h	SUB A,expr	Immediate		c z	4
12h	SUB A,[expr]	Direct Address		c z	6
13h	SUB A,[X+expr]	Index		c z	7
14h	SUB [expr],A	Direct Address		c z	7
15h	SUB [X+expr],A	Index		c z	8
16h	SUB [expr],expr	Direct Address	Immediate	c z	9
17h	SUB [X+expr],expr	Index	Immediate	c z	10
DEC		Decrement			
78h	DEC A			c z	4
79h	DEC X			c z	4
7Ah	DEC [expr]	Direct Address		c z	7
7Bh	DEC [X+expr]	Index		c z	8
INC		Increment			
74h	INC A			c z	4
75h	INC X			c z	4
76h	INC [expr]	Direct Address		c z	7
77h	INC [X+expr]	Index		c z	8
ASL		Arithmetic Shift Left			
64h	ASL A			c z	4
65h	ASL [expr]	Direct Address		c z	7
66h	ASL [X+expr]	Index		c z	8
ASR		Arithmetic Shift Right			
67h	ASR A			c z	4
68h	ASR [expr]	Direct Address		c z	7
69h	ASR [X+expr]	Index		c z	8

Movement Instructions				Flags	Cycles
INDEX		Table Read (relative)			
Fxh	INDEX addr	LSB Address Byte (MSN in opcode, x)		z	13
MOV		Move			
4Fh	MOV X,SP				4
50h	MOV A,expr	Immediate		z	4
51h	MOV A,[expr]	Direct Address		z	5
52h	MOV A,[X+expr]	Index		z	6
53h	MOV [expr],A	Direct Address			5
54h	MOV [X+expr],A	Index			6
55h	MOV [expr],expr	Direct Address	Immediate		8
56h	MOV [X+expr],expr	Index	Immediate		9
57h	MOV X,expr	Immediate			4
58h	MOV X,[expr]	Direct Address			6
59h	MOV X,[X+expr]	Index			7
5Ah	MOV [expr],X	Direct Address			5
5Bh	MOV A,X			z	4
5Ch	MOV X,A				4
5Dh	MOV A,REG[expr]	Reg Direct Address		z	6
5Eh	MOV A,REG[X+expr]	Reg Index		z	7
5Fh	MOV [expr],[expr]	Direct Address	Direct Address		7
60h	MOV REG[expr],A	Reg Direct Address			5
61h	MOV REG[X+expr],A	Reg Index			6
62h	MOV REG[expr],expr	Reg Direct Address	Immediate		8
63h	MOV REG[X+expr],expr	Reg Index	Immediate		9
MVI		Move Indirect, Post Increment to Memory			
3Eh	MVI A,[expr]	Direct Address (Page 0)		z	10
3Fh	MVI [expr],A	Direct Address (Page 0)			10
POP		Pop Stack into Register			
18h	POP A			z	5
20h	POP X				5
PUSH		Push Register onto Stack			
08h	PUSH A				4
10h	PUSH X				4
ROMX		Table Read		z	11
SWAP		Swap			
4Bh	SWAP A,X			z	5
4Ch	SWAP A,[expr]	Direct Address		z	7
4Dh	SWAP X,[expr]	Direct Address		z	7
4Eh	SWAP A,SP			z	5

Logical Instructions				Flags	Cycles
AND		Bitwise AND			
21h	AND A,expr	Immediate		z	4
22h	AND A,[expr]	Direct Address		z	6
23h	AND A,[X+expr]	Index		z	7
24h	AND [expr],A	Direct Address		z	7
25h	AND [X+expr],A	Index		z	8
26h	AND [expr],expr	Direct Address	Immediate	z	9
27h	AND [X+expr],expr	Index	Immediate	z	10
41h	AND REG[expr],expr	Reg Direct Address	Immediate	z	9
42h	AND REG[X+expr],expr	Reg Index	Immediate	z	10
70h	AND F,expr	Immediate		c z	4
OR		Bitwise OR			
29h	OR A,expr	Immediate		z	4
2Ah	OR A,[expr]	Direct Address		z	6
2Bh	OR A,[X+expr]	Index		z	7
2Ch	OR [expr],A	Direct Address		z	7
2Dh	OR [X+expr],A	Index		z	8
2Eh	OR [expr],expr	Direct Address	Immediate	z	9
2Fh	OR [X+expr],expr	Index	Immediate	z	10
43h	OR REG[expr],expr	Reg Direct Address	Immediate	z	9
44h	OR REG[X+expr],expr	Reg Index	Immediate	z	10
71h	OR F,expr	Immediate		c z	4
XOR		Bitwise XOR			
31h	XOR A,expr	Immediate		z	4
32h	XOR A,[expr]	Direct Address		z	6
33h	XOR A,[X+expr]	Index		z	7
34h	XOR [expr],A	Direct Address		z	7
35h	XOR [X+expr],A	Index		z	8
36h	XOR [expr],expr	Direct Address	Immediate	z	9
37h	XOR [X+expr],expr	Index	Immediate	z	10
45h	XOR REG[expr],expr	Reg Direct Address	Immediate	z	9
46h	XOR REG[X+expr],expr	Reg Index	Immediate	z	10
72h	XOR F,expr	Immediate		c z	4
CPL		Complement Accumulator			
73h	CPL A			z	4
RLC		Rotate Left through Carry			
6Ah	RLC A			c z	4
6Bh	RLC [expr]	Direct Address		c z	7
6Ch	RLC [X+expr]	Index		c z	8
RRC		Rotate Right through Carry			
6Dh	RRC A			c z	4
6Eh	RRC [expr]	Direct Address		c z	7
6Fh	RRC [X+expr]	Index		c z	8

Appendix A: Application Interface Notes

Interfacing C and Assembly

To optimize argument passing and return value activities between the PSoC Designer C Compiler and Assembler, employ the `#pragma fastcall`.

The fastcall convention was devised to create an efficient argument/return value mechanism between 'C' and assembly language functions.

Fastcall is only used by 'C' functions calling assembly written functions. Functions written in 'C' cannot utilize the fastcall convention.

The following table reflects the set of `#pragma fastcall` conventions used for *argument passing* register assignments:

Argument Type	Argument Register	Comment
char	A	
char, char	A, X	First char in A and second in X
int	X, A	MSB in X and LSB in A
Pointer	A, X	MSB in A and LSB in X
char, ...	A, X	First argument passed in A. Successive arguments are pointed to by X, where X is set up as a pointer to the remaining arguments. Typically, these arguments are stored on the stack
Int,...	X	X is set up as a pointer that points to the contiguous block of memory that stores the arguments. Typically, the arguments are stored on the stack.
All the others	X	Same as above

Arguments that are pushed on the stack are pushed from right to left.

The reference of returned structures reside in the A and X registers. If passed by value, a structure is always passed through the stack, and not in registers. Passing a structure by reference (i.e., passing the address of a structure) is the same as passing the address of any data item, that is, a pointer (which is 2 bytes).

The following table reflects the set of `#pragma fastcall` conventions used for *return value* register assignments:

Return Type	Return Register	Comment
char	A	
int	X, A	
long	__r0..__r3	Delivered in the virtual registers
pointer	A, X	

Index

Accessing the Assembler	11	LJMP	56
ADC	40	Macro Definition Start - MACRO/Macro	
ADD	41	Definition End – ENDMACRO	36
Address Spaces	6, 14	Menu Options	12
Addressing Modes	6, 16	MOV	57
Alternative Result of IF...ELSE...ENDIF -		MVI	58
ELSE	34	NOP	59
AND	42	Notation Standards	4, 6
Appendix A: Application Interface Notes...	78	Opening PSoC Designer	11
Area - AREA	29	OR	60
Area Origin - ORG	37	POP	61
ASL	43	Product Upgrades	10
ASR	43	Purpose	9
CALL	44	PUSH	61
CMP	45	RAM Block - BLK	31
CPL	45	RAM Block in Words - BLKW	31
DEC	46	RET	62
Define ASCII String - DS	32	RETI	62
Define Byte - DB	32	RLC	63
Define UNICODE String - DSU	33	ROMX	63
Define Word - DW	33	RRC	64
Define Word, Little Endian Ordering - DWL		SBB	65
.....	34	Section 1. Introduction	6, 9
Destination of Instruction Results	6, 22	Section 2. Accessing the Assembler	6, 11
Documentation Conventions	3, 6	Section 3. The Microprocessor	6, 13
Equate Label - EQU	35	Section 4. Assembly File Syntax	6, 23
Export - EXPORT	35	Section 5. List File Format	6, 27
HALT	47	Section 6. Assembler Directives	7, 29
IF...ELSE...ENDIF - ENDIF	34	Section 7. Instruction Set	7, 39
IF...ELSE...ENDIF - IF	35	Section 8. Compile/Assemble Error	
INC	48	Messages	8, 71
Include Source File - INCLUDE	36	Section Overview	9
INDEX	49	SSC	68
Instruction Format	6, 15	SUB	66
Instruction Set Reference Table	8, 77	Support	10
JACC	50	SWAP	67
JC	51	Syntax	6, 23
JMP	52	Syntax Details	6
JNC	53	Syntax Details	23
JNZ	54	TST	69
JZ55		Two-Minute Overview	2, 6
LCALL	56	XOR	70