



ImageCraft Assembly Language Guide

Document # 001-44475 Rev. *B

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone (USA): 800.858.1810
Phone (Intl): 408.943.2600
<http://www.cypress.com>

Copyrights

Copyright © 2001 - 2011 Cypress Semiconductor Corporation. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC Designer™, Programmable System-on-Chip™ is trademarks and PSoC® is a registered trademark of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.

Contents



1. Introduction	7
1.1 Chapter Overviews	7
1.2 Support	8
1.2.1 Technical Support Systems	8
1.2.2 Product Upgrades	8
1.3 Documentation Conventions	8
1.4 Revision History	9
2. M8C Microprocessor	11
2.1 Internal Registers	11
2.2 Address Spaces	12
2.3 Instruction Set Summary	14
2.4 Instruction Formats	16
2.4.1 One-Byte Instruction	16
2.4.2 Two-Byte Instructions	16
2.4.3 Three-Byte Instructions	17
2.5 Addressing Modes	18
2.5.1 Source Immediate	18
2.5.2 Source Direct	19
2.5.3 Source Indexed	19
2.5.4 Destination Direct	20
2.5.5 Destination Indexed	20
2.5.6 Destination Direct Source Immediate	21
2.5.7 Destination Indexed Source Immediate	21
2.5.8 Destination Direct Source Direct	22
2.5.9 Source Indirect Post Increment	22
2.5.10 Destination Indirect Post Increment	23
3. ImageCraft Assembler	25
3.1 Source File Format	25
3.1.1 Labels	26
3.1.2 Mnemonics	27
3.1.3 Operands	28
3.1.4 Comments	29
3.1.5 Directives	30
3.2 Listing File Format	30
3.3 Map File Format	30
3.4 ROM File Format	30
3.5 Intel® HEX File Format	32
3.6 Convention for Restoring Internal Registers	34
3.7 Compiling a File into a Library Module	34

4. M8C Instruction Set	37
Add with Carry	ADC 38
Add without Carry	ADD 39
Bitwise AND	AND 40
Arithmetic Shift Left	ASL 41
Arithmetic Shift Right	ASR 42
Call Function	CALL 43
Non-Destructive Compare	CMP 44
Complement Accumulator	CPL 45
Decrement	DEC 46
Halt	HALT 47
Increment	INC 48
Relative Table Read	INDEX 49
Jump Accumulator	JACC 50
Jump if Carry	JC 51
Jump	JMP 52
Jump if No Carry	JNC 53
Jump if Not Zero	JNZ 54
Jump if Zero	JZ 55
Long Call	LCALL 56
Long Jump	LJMP 57
Move	MOV 58
Move Indirect, Post-Increment to Memory	MVI 59
No Operation	NOP 60
Bitwise OR	OR 61
Pop Stack into Register	POP 62
Push Register onto Stack	PUSH 63
Return	RET 64
Return from Interrupt	RETI 65
Rotate Left through Carry	RLC 66
Absolute Table Read	ROMX 67
Rotate Right through Carry	RRC 68
Subtract with Borrow	SBB 69
Subtract without Borrow	SUB 70
Swap	SWAP 71
System Supervisor Call	SSC 72
Test for Mask	TST 73
Bitwise XOR	XOR 74
 5. Assembler Directives	 75
Area	AREA 76
5.1.1 Code Compressor and the AREA Directive	77
NULL Terminated ASCII String	ASCIZ 78
RAM Block in Bytes	BLK 79
RAM Block in Words	BLKW 80
Define Byte	DB 81
Define Floating-point Number	DF 82
Define ASCII String	DS 83
Define UNICODE String	DSU 84
Define Word, Big Endian Ordering	DW 85
Define Word, Little Endian Ordering	DWL 86
Equate Label	EQU 87

Export	EXPORT 88
Conditional Source	IF, ELSE, ENDIF 89
Include Source File	INCLUDE 90
Prevent Code Compression of DataLITERAL, .ENDLITERAL 91
Macro Definition	MACRO, ENDM 92
Area Origin	ORG 93
Section for Dead-Code EliminationSECTION, .ENDSECTION 94
Suspend/Resume Code CompressorOR F,0; ADD SP,0 94

6. Large Memory Model	97
6.1 Large Memory Model Registers	97
6.1.1 Current Page Pointer (CUR_PP)	97
6.1.2 Stack Page Pointer (STK_PP)	98
6.1.3 Index Page Pointer (IDX_PP)	98
6.1.4 MVI Page Pointers (MVR_PP, MVW_PP)	98
6.2 Large Memory Model Firmware	99
6.2.1 Memory.inc	99
6.2.2 Native Paging Mode	99
6.2.3 LMM Register Policies	100
6.2.4 API Classes and Page Pointer Usage	100
6.2.5 Fastcall16	101
6.3 Going from Small to Large Memory Model	102
6.3.1 Example 1 - Simple Data Movement	102
6.3.2 Example 2 - Moving Stack Data	103
6.3.3 Example 3 - Array Access	105
6.3.4 Example 4 - MVI Instructions	106
6.4 Calling Functions	108
6.5 Interrupt Service Routines	109
6.6 Reducing Memory Usage in LMM	110
6.6.1 Create a single memory structure and use constants to index into the structure	110
6.6.2 Use the InterruptRAM AREA declaration to put variables on Page 0	111
6.6.3 Create unique RAM AREAs and force the linker to put each on the same page	112
6.6.4 Forcing RAM Areas to Specific Pages in C	113
6.7 LMM Macros	113
6.7.1 RAM_SETPAGE_CUR (PG_NUMBER)	113
6.7.2 RAM_SETPAGE_STK (PG_NUMBER)	113
6.7.3 RAM_SETPAGE_IDX (PG_NUMBER)	113
6.7.4 RAM_SETPAGE_MVR (PG_NUMBER)	113
6.7.5 RAM_SETPAGE_MVW (PG_NUMBER)	114
6.7.6 RAM_SETPAGE_IDX2STK	114
6.7.7 RAM_CHANGE_PAGE_MODE (MODE)	114
6.7.8 RAM_RESTORE_NATIVE_PAGING	114
6.7.9 RAM_SET_NATIVE_PAGING	114
6.7.10 RAM_X_POINTS_TO_STACKPAGE	114
6.7.11 RAM_X_POINTS_TO_INDEXPAGE	115
6.7.12 REG_PRESERVE (IOReg)	115
6.7.13 REG_PRESERVE (IOReg)	115
6.7.14 ISR_PRESERVE_PAGE_POINTERS	115
6.7.15 ISR_RESTORE_PAGE_POINTERS	115

7. Builds and Error Messages	117
7.1 Assemble and Build	117
7.2 Linker Operations	117
7.3 Code Compressor and Dead-Code Elimination Error Messages	118
 A. Reference Tables Appendix	 119
A.1 Assembly Syntax Expressions	119
A.2 Operand Constant Formats	119
A.3 Assembler Directives Summary	120
A.4 ASCII Code Table.....	121
A.5 Instruction Set Summary	122
 Index	 125

1. Introduction



The PSoC Designer Assembly Language Guide documents the assembly language instruction set for the M8C microcontroller as well as other compatible assembly practices. It covers the ImageCraft Assembler.

The PSoC Designer Integrated Development Environment (IDE) software is available free of charge and supports development in assembly language. For customers interested in developing in C, compilers are available. Please contact your local distributor if you are interested in purchasing a C Compiler for PSoC Designer. For more information about developing in C for the PSoC device, please read the *PSoC Designer C Language Compiler Guide* available at the Cypress web site at www.cypress.com.

1.1 Chapter Overviews

Table 1-1. Overview of the Assembly Language Guide

Chapter	Description
Introduction (on page 7)	Describes the purpose of this guide, overviews each chapter, supplies product support and upgrade information, and lists documentation conventions.
M8C Microprocessor (on page 11)	Discusses the microprocessor and explains address spaces, instruction format, and destination of instruction results. It also lists all addressing modes and provides examples of each.
ImageCraft Assembler (on page 25)	Provides assembly language source syntax including labels, mnemonics, operands, comments, and directives. Describes the various file formats created by the ImageCraft Assembler, along with the convention for restoring internal registers and compiling a file into a library module.
M8C Instruction Set (on page 37)	Provides a detailed list of all M8C instructions. Information about individual M8C instructions is also available via <i>PSoC Designer Online Help</i> .
Assembler Directives (on page 75)	Provides a detailed list of all ImageCraft Assembler directives.
Large Memory Model chapter on page 97	This chapter presents a detailed overview of the memory architecture designed to accommodate PSoC® 1 devices with a RAM space larger than 256 bytes.
Builds and Error Messages (on page 117)	Supplies several lists of assembler-related errors and warnings, along with their possible solutions.
Appendix A Reference Tables Appendix (on page 119)	Serves as a quick reference to the M8C instruction set, and assembler directives and syntax expressions, along with an ASCII code table.

1.2 Support

Free support for PSoC Designer is available online, just click on PSoC Mixed-Signal Controllers then Technical Support. Resources include Training Seminars, Discussion Forums, Application Notes, PSoC Consultants, TightLink Technical Support Email/Knowledge Base, and Application Support Technicians.

Before utilizing the Cypress support services, know the version of PSoC Designer installed on your system. To quickly determine the version, build, or service pack of your current installation of PSoC Designer, click Help > About PSoC Designer.

Support for the ImageCraft C Compiler and Assembler is available from ImageCraft.

<http://www.imagecraft.com/>

1.2.1 Technical Support Systems

Enter a technical support request in this system with a guaranteed response time of four hours at

<http://www.cypress.com/support/login.cfm>

1.2.2 Product Upgrades

Cypress provides scheduled upgrades and version enhancements for PSoC software free of charge. You can order upgrades from your distributor on CD-ROM or download them directly from www.cypress.com under Software and Drivers. Critical updates to system documentation are also available on the Cypress web site.

1.3 Documentation Conventions

The following are easily identifiable conventions used throughout this guide.

Table 1-2. Documentation Conventions

Convention	Usage
Courier New	Displays file locations, user entered text, and source code: C:\...cd\icc\
<i>Italics</i>	Displays file names and reference documentation: Read about the <i>sourcefile.hex</i> file in the <i>PSoC Designer Guide</i> .
[Bracketed, Bold]	Displays keyboard commands in procedures: [Enter] or [Ctrl] [C]
File > Open	Represents menu paths: File > Open > New Project
Bold	Displays commands, menu paths, and icon names in procedures: Click the File icon and then click Open .
Text in gray boxes	Presents cautions or unique functionality of the product.

The following are acronyms used throughout this guide.

Table 1-3. Acronyms

Acronym	Description
A	CPU_A register (accumulator)
CF	carry flag
F	CPU_F register (flags ZF, CF, and others)
GIE	global enable interrupt
IDE	integrated development environment

Table 1-3. Acronyms

Acronym	Description
NOP	no operation
PC	CPU_PC register (program counter)
POR	power-on-reset
RAM	random access memory
REG	register space
ROM	read only memory
SP	CPU_SP register (stack pointer)
SROM	supervisory read only memory
SSC	supervisory system call
WDR	watchdog timer reset
X	CPU_X register (index)
XRES	external reset
ZF	zero flag

1.4 Revision History

Table 1-4. Revision History

Revision	PDF Creation Date	Origin of Change	Description of Change
**	May 20, 2008	FSU	Put the original ImageCraft Assembly Guide in a new template and assigned a Spec Number.
*A	Sept 11, 2008	OGNE/ PYRS	Made changes in the Assembly instructions chapter, page 72.
*B	December 01, 2011	RAVG/ECI	Made changes in the Assembly Assembler Introduction. And added Note under section 3.1.1 - Labels Section 5.1 added and updated. New note added below section 5.1. Section 5.1.1 Code Compressor and the AREA Directive - Update Chapter 6 - Large Memory Model Programming added.

2. M8C Microprocessor



This chapter covers internal M8C registers, address spaces, instruction summary and formats, and addressing modes for the M8C microprocessor. The M8C is a 4 MIPS 8-bit Harvard architecture microprocessor. Code selectable processor clock speeds from 93.7 kHz to 24 MHz allow the M8C to be tuned to a particular application's performance and power requirements. The M8C supports a rich instruction set which allows for efficient low-level language support. For a detailed description of all M8C instructions, refer to the [M8C Instruction Set chapter on page 37](#).

2.1 Internal Registers

The M8C has five internal registers that are used in program execution:

- Accumulator (A)
- Index (X)
- Program Counter (PC)
- Stack Pointer (SP)
- Flags (F)

All of the internal M8C registers are 8 bits in width, except for the PC (CPU_PC register) which is 16 bits wide. Upon reset, A, X, PC, and SP are reset to 0x00. The Flag register CPU_F (F) is reset to 0x02 indicating that the Z flag is set.

With each stack operation, the SP is automatically incremented or decremented so that it always points at the next stack byte in Random Access Memory (RAM). If the last byte in the stack is at address 0xFF in RAM, the Stack Pointer (CPU_SP or SP) will wrap to RAM address 0x00. It is the firmware developer's responsibility to ensure that the stack does not overlap with user-defined variables in RAM.

As shown in [Table 2-1](#), the Flag register has 6 of 8 bits defined. The PgMode and XIO bits are used to control the active register and RAM address spaces in the PSoC device. The C and Z bits are the Carry and Zero flags respectively. These flags are affected by arithmetic, logical, and shift operations provided in the M8C instruction. The GIE bit is the Global Interrupt Enable. When set, this bit allows the M8C to be interrupted by the PSoC device's interrupt controller.

Table 2-1. M8C Internal Flag (F) Register (CPU_F)

Bits	7	6	5	4	3	2	1	0
Name	PgMode[1:0]			XIO		C	Z	GIE

With the exception of the CPU_F register, the M8C internal registers are not accessible via an explicit register address. PSoC parts in the CY8C25xxx and CY8C26xxx device family do not have a readable CPU_F register. The `OR F, expr` and `AND F, expr` instructions must be used to set and clear CPU_F register bits. The internal M8C registers are accessed using special instructions such as:

- `MOV A, expr`
- `MOV X, expr`
- `SWAP A, SP`
- `OR F, expr`
- `JMP`

The CPU_F register may be read by using address `0xF7` in any register bank, except in CY8C25xxx and CY8C26xxx devices.

2.2 Address Spaces

The M8C microcontroller has three address spaces: ROM, RAM, and registers. The Read Only Memory (ROM) address space is accessed via its own address and data bus. [Figure 2-1](#) illustrates the arrangement of the PSoC device address spaces.

The ROM address space is composed of the Supervisory ROM and the on-chip Flash program store. Flash is organized into 64-byte blocks. The user need not be concerned with program store page boundaries, because the M8C automatically increments the 16-bit CPU_PC register (PC) on every instruction making the block boundaries invisible to user code. Instructions occurring on a 256-byte Flash page boundary (with the exception of jump instructions) incur an extra M8C clock cycle because the upper byte of the Program Counter (PC) is incremented.

The register address space is used to configure the PSoC device's programmable blocks. It consists of two banks of 256 bytes each. To switch between banks, the XIO bit in the Flag register is set or cleared (set for Bank1 = Configuration Space, cleared for Bank0 = User Space). The common convention is to leave the bank set to Bank0 (XIO cleared), switch to Bank1 as needed (set XIO), then switch back to Bank0.

RAM is broken into 256-byte pages. For PSoC devices with 256 bytes of RAM or less, the program stack is stored in RAM Page 0. For PSoC devices with 512 bytes of RAM or more, the stack is constrained to the last RAM page. For information on RAM configuration in a specific device, refer to the device-specific data sheet.

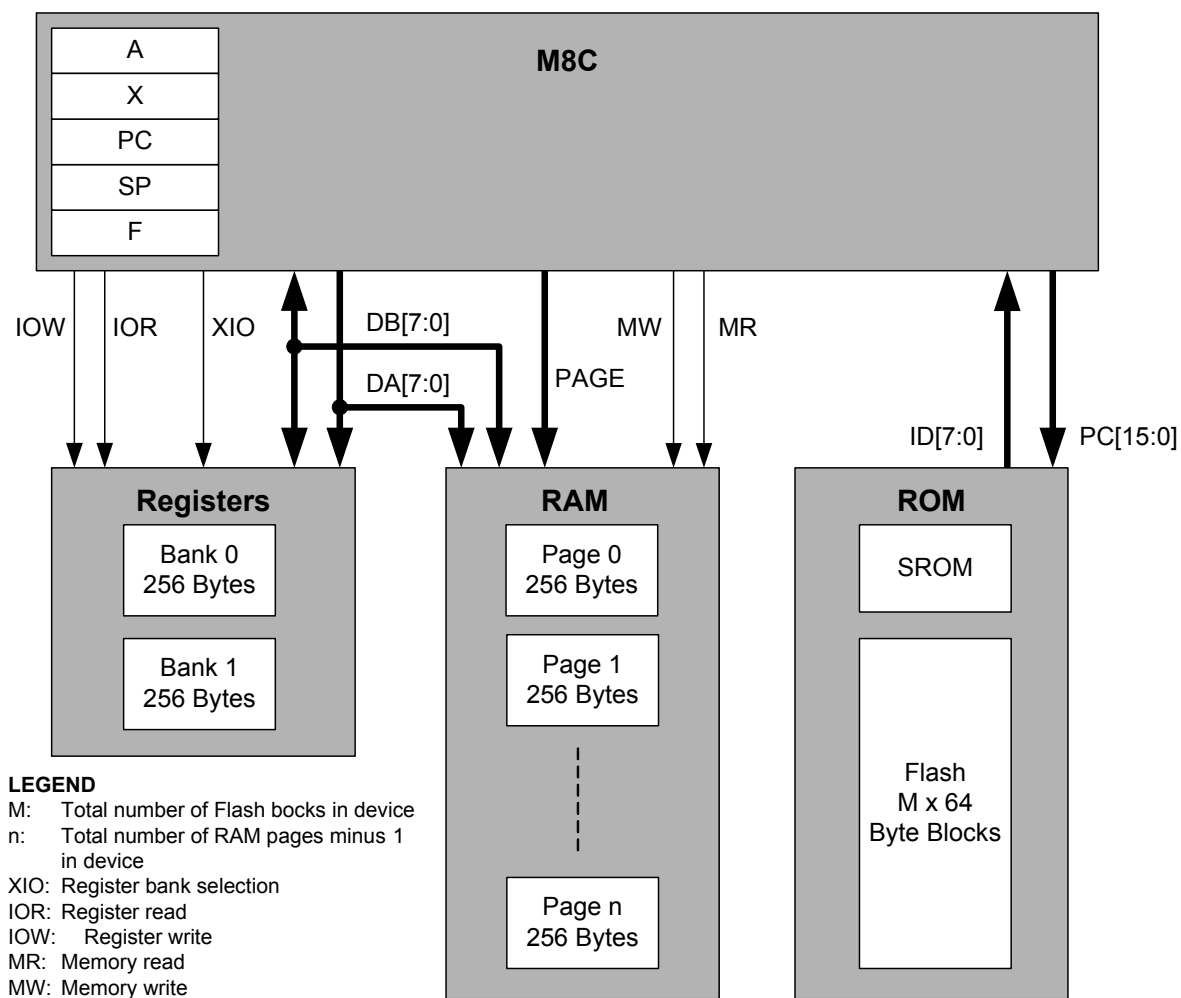


Figure 2-1. M8C Microcontroller Address Spaces

2.3 Instruction Set Summary

The instruction set is summarized in both [Table 2-2](#) and [Table 2-3](#) (in numeric and mnemonic order, respectively), and serves as a quick reference.

Table 2-2. Instruction Set Summary Sorted Numerically by Opcode

Opcode Hex	Cycles	Bytes	Instruction Format	Flags	Opcode Hex	Cycles	Bytes	Instruction Format	Flags	Opcode Hex	Cycles	Bytes	Instruction Format	Flags
00	15	1	SSC		2D	8	2	OR [X+expr], A	Z	5A	5	2	MOV [expr], X	
01	4	2	ADD A, expr	C, Z	2E	9	3	OR [expr], expr	Z	5B	4	1	MOV A, X	Z
02	6	2	ADD A, [expr]	C, Z	2F	10	3	OR [X+expr], expr	Z	5C	4	1	MOV X, A	
03	7	2	ADD A, [X+expr]	C, Z	30	9	1	HALT		5D	6	2	MOV A, reg[expr]	Z
04	7	2	ADD [expr], A	C, Z	31	4	2	XOR A, expr	Z	5E	7	2	MOV A, reg[X+expr]	Z
05	8	2	ADD [X+expr], A	C, Z	32	6	2	XOR A, [expr]	Z	5F	10	3	MOV [expr], [expr]	
06	9	3	ADD [expr], expr	C, Z	33	7	2	XOR A, [X+expr]	Z	60	5	2	MOV reg[expr], A	
07	10	3	ADD [X+expr], expr	C, Z	34	7	2	XOR [expr], A	Z	61	6	2	MOV reg[X+expr], A	
08	4	1	PUSH A		35	8	2	XOR [X+expr], A	Z	62	8	3	MOV reg[expr], expr	
09	4	2	ADC A, expr	C, Z	36	9	3	XOR [expr], expr	Z	63	9	3	MOV reg[X+expr], expr	
0A	6	2	ADC A, [expr]	C, Z	37	10	3	XOR [X+expr], expr	Z	64	4	1	ASL A	C, Z
0B	7	2	ADC A, [X+expr]	C, Z	38	5	2	ADD SP, expr		65	7	2	ASL [expr]	C, Z
0C	7	2	ADC [expr], A	C, Z	39	5	2	CMP A, expr		66	8	2	ASL [X+expr]	C, Z
0D	8	2	ADC [X+expr], A	C, Z	3A	7	2	CMP A, [expr]	if (A=B) Z=1 if (A<B) C=1	67	4	1	ASR A	C, Z
0E	9	3	ADC [expr], expr	C, Z	3B	8	2	CMP A, [X+expr]		68	7	2	ASR [expr]	C, Z
0F	10	3	ADC [X+expr], expr	C, Z	3C	8	3	CMP [expr], expr		69	8	2	ASR [X+expr]	C, Z
10	4	1	PUSH X		3D	9	3	CMP [X+expr], expr		6A	4	1	RLC A	C, Z
11	4	2	SUB A, expr	C, Z	3E	10	2	MVI A, [[expr]++]	Z	6B	7	2	RLC [expr]	C, Z
12	6	2	SUB A, [expr]	C, Z	3F	10	2	MVI [[expr]++], A		6C	8	2	RLC [X+expr]	C, Z
13	7	2	SUB A, [X+expr]	C, Z	40	4	1	NOP		6D	4	1	RRC A	C, Z
14	7	2	SUB [expr], A	C, Z	41	9	3	AND reg[expr], expr	Z	6E	7	2	RRC [expr]	C, Z
15	8	2	SUB [X+expr], A	C, Z	42	10	3	AND reg[X+expr], expr	Z	6F	8	2	RRC [X+expr]	C, Z
16	9	3	SUB [expr], expr	C, Z	43	9	3	OR reg[expr], expr	Z	70	4	2	AND F, expr	C, Z
17	10	3	SUB [X+expr], expr	C, Z	44	10	3	OR reg[X+expr], expr	Z	71	4	2	OR F, expr	C, Z
18	5	1	POP A	Z	45	9	3	XOR reg[expr], expr	Z	72	4	2	XOR F, expr	C, Z
19	4	2	SBB A, expr	C, Z	46	10	3	XOR reg[X+expr], expr	Z	73	4	1	CPL A	Z
1A	6	2	SBB A, [expr]	C, Z	47	8	3	TST [expr], expr	Z	74	4	1	INC A	C, Z
1B	7	2	SBB A, [X+expr]	C, Z	48	9	3	TST [X+expr], expr	Z	75	4	1	INC X	C, Z
1C	7	2	SBB [expr], A	C, Z	49	9	3	TST reg[expr], expr	Z	76	7	2	INC [expr]	C, Z
1D	8	2	SBB [X+expr], A	C, Z	4A	10	3	TST reg[X+expr], expr	Z	77	8	2	INC [X+expr]	C, Z
1E	9	3	SBB [expr], expr	C, Z	4B	5	1	SWAP A, X	Z	78	4	1	DEC A	C, Z
1F	10	3	SBB [X+expr], expr	C, Z	4C	7	2	SWAP A, [expr]	Z	79	4	1	DEC X	C, Z
20	5	1	POP X		4D	7	2	SWAP X, [expr]		7A	7	2	DEC [expr]	C, Z
21	4	2	AND A, expr	Z	4E	5	1	SWAP A, SP	Z	7B	8	2	DEC [X+expr]	C, Z
22	6	2	AND A, [expr]	Z	4F	4	1	MOV X, SP		7C	13	3	LCALL	
23	7	2	AND A, [X+expr]	Z	50	4	2	MOV A, expr	Z	7D	7	3	LJMP	
24	7	2	AND [expr], A	Z	51	5	2	MOV A, [expr]	Z	7E	10	1	RETI	C, Z
25	8	2	AND [X+expr], A	Z	52	6	2	MOV A, [X+expr]	Z	7F	8	1	RET	
26	9	3	AND [expr], expr	Z	53	5	2	MOV [expr], A		8x	5	2	JMP	
27	10	3	AND [X+expr], expr	Z	54	6	2	MOV [X+expr], A		9x	11	2	CALL	
28	11	1	ROMX	Z	55	8	3	MOV [expr], expr		Ax	5	2	JZ	
29	4	2	OR A, expr	Z	56	9	3	MOV [X+expr], expr		Bx	5	2	JNZ	
2A	6	2	OR A, [expr]	Z	57	4	2	MOV X, expr		Cx	5	2	JC	
2B	7	2	OR A, [X+expr]	Z	58	6	2	MOV X, [expr]		Dx	5	2	JNC	
2C	7	2	OR [expr], A	Z	59	7	2	MOV X, [X+expr]		Ex	7	2	JACC	
										Fx	13	2	INDEX	Z

Note 1 Interrupt acknowledge to Interrupt Vector table = 13 cycles.

Note 2 The number of cycles required by an instruction is increased by one for instructions that span 256 byte page boundaries in the Flash memory space.

Table 2-3. Instruction Set Summary Sorted Alphabetically by Mnemonic

Opcode Hex	Cycles	Bytes	Instruction Format	Flags	Opcode Hex	Cycles	Bytes	Instruction Format	Flags	Opcode Hex	Cycles	Bytes	Instruction Format	Flags
09	4	2	ADC A, expr	C, Z	76	7	2	INC [expr]	C, Z	20	5	1	POP X	
0A	6	2	ADC A, [expr]	C, Z	77	8	2	INC [X+expr]	C, Z	18	5	1	POP A	Z
0B	7	2	ADC A, [X+expr]	C, Z	Fx	13	2	INDEX	Z	10	4	1	PUSH X	
0C	7	2	ADC [expr], A	C, Z	Ex	7	2	JACC		08	4	1	PUSH A	
0D	8	2	ADC [X+expr], A	C, Z	Cx	5	2	JC		7E	10	1	RETI	C, Z
0E	9	3	ADC [expr], expr	C, Z	8x	5	2	JMP		7F	8	1	RET	
0F	10	3	ADC [X+expr], expr	C, Z	Dx	5	2	JNC		6A	4	1	RLC A	C, Z
01	4	2	ADD A, expr	C, Z	Bx	5	2	JNZ		6B	7	2	RLC [expr]	C, Z
02	6	2	ADD A, [expr]	C, Z	Ax	5	2	JZ		6C	8	2	RLC [X+expr]	C, Z
03	7	2	ADD A, [X+expr]	C, Z	7C	13	3	LCALL		28	11	1	ROMX	Z
04	7	2	ADD [expr], A	C, Z	7D	7	3	LJMP		6D	4	1	RRC A	C, Z
05	8	2	ADD [X+expr], A	C, Z	4F	4	1	MOV X, SP		6E	7	2	RRC [expr]	C, Z
06	9	3	ADD [expr], expr	C, Z	50	4	2	MOV A, expr	Z	6F	8	2	RRC [X+expr]	C, Z
07	10	3	ADD [X+expr], expr	C, Z	51	5	2	MOV A, [expr]	Z	19	4	2	SBB A, expr	C, Z
38	5	2	ADD SP, expr		52	6	2	MOV A, [X+expr]	Z	1A	6	2	SBB A, [expr]	C, Z
21	4	2	AND A, expr	Z	53	5	2	MOV [expr], A		1B	7	2	SBB A, [X+expr]	C, Z
22	6	2	AND A, [expr]	Z	54	6	2	MOV [X+expr], A		1C	7	2	SBB [expr], A	C, Z
23	7	2	AND A, [X+expr]	Z	55	8	3	MOV [expr], expr		1D	8	2	SBB [X+expr], A	C, Z
24	7	2	AND [expr], A	Z	56	9	3	MOV [X+expr], expr		1E	9	3	SBB [expr], expr	C, Z
25	8	2	AND [X+expr], A	Z	57	4	2	MOV X, expr		1F	10	3	SBB [X+expr], expr	C, Z
26	9	3	AND [expr], expr	Z	58	6	2	MOV X, [expr]		00	15	1	SSC	
27	10	3	AND [X+expr], expr	Z	59	7	2	MOV X, [X+expr]		11	4	2	SUB A, expr	C, Z
70	4	2	AND F, expr	C, Z	5A	5	2	MOV [expr], X		12	6	2	SUB A, [expr]	C, Z
41	9	3	AND reg[expr], expr	Z	5B	4	1	MOV A, X	Z	13	7	2	SUB A, [X+expr]	C, Z
42	10	3	AND reg[X+expr], expr	Z	5C	4	1	MOV X, A		14	7	2	SUB [expr], A	C, Z
64	4	1	ASL A	C, Z	5D	6	2	MOV A, reg[expr]	Z	15	8	2	SUB [X+expr], A	C, Z
65	7	2	ASL [expr]	C, Z	5E	7	2	MOV A, reg[X+expr]	Z	16	9	3	SUB [expr], expr	C, Z
66	8	2	ASL [X+expr]	C, Z	5F	10	3	MOV [expr], [expr]		17	10	3	SUB [X+expr], expr	C, Z
67	4	1	ASR A	C, Z	60	5	2	MOV reg[expr], A		4B	5	1	SWAP A, X	Z
68	7	2	ASR [expr]	C, Z	61	6	2	MOV reg[X+expr], A		4C	7	2	SWAP A, [expr]	Z
69	8	2	ASR [X+expr]	C, Z	62	8	3	MOV reg[expr], expr		4D	7	2	SWAP X, [expr]	
9x	11	2	CALL		63	9	3	MOV reg[X+expr], expr		4E	5	1	SWAP A, SP	Z
39	5	2	CMP A, expr		3E	10	2	MVI A, [[expr]++]	Z	47	8	3	TST [expr], expr	Z
3A	7	2	CMP A, [expr]	if (A=B) Z=1	3F	10	2	MVI [[expr]++], A		48	9	3	TST [X+expr], expr	Z
3B	8	2	CMP A, [X+expr]		40	4	1	NOP		49	9	3	TST reg[expr], expr	Z
3C	8	3	CMP [expr], expr	if (A<B) C=1	29	4	2	OR A, expr	Z	4A	10	3	TST reg[X+expr], expr	Z
3D	9	3	CMP [X+expr], expr		2A	6	2	OR A, [expr]	Z	72	4	2	XOR F, expr	C, Z
73	4	1	CPL A	Z	2B	7	2	OR A, [X+expr]	Z	31	4	2	XOR A, expr	Z
78	4	1	DEC A	C, Z	2C	7	2	OR [expr], A	Z	32	6	2	XOR A, [expr]	Z
79	4	1	DEC X	C, Z	2D	8	2	OR [X+expr], A	Z	33	7	2	XOR A, [X+expr]	Z
7A	7	2	DEC [expr]	C, Z	2E	9	3	OR [expr], expr	Z	34	7	2	XOR [expr], A	Z
7B	8	2	DEC [X+expr]	C, Z	2F	10	3	OR [X+expr], expr	Z	35	8	2	XOR [X+expr], A	Z
30	9	1	HALT		43	9	3	OR reg[expr], expr	Z	36	9	3	XOR [expr], expr	Z
74	4	1	INC A	C, Z	44	10	3	OR reg[X+expr], expr	Z	37	10	3	XOR [X+expr], expr	Z
75	4	1	INC X	C, Z	71	4	2	OR F, expr	C, Z	45	9	3	XOR reg[expr], expr	Z
										46	10	3	XOR reg[X+expr], expr	Z

Note 1 Interrupt acknowledge to Interrupt Vector table = 13 cycles.

Note 2 The number of cycles required by an instruction is increased by one for instructions that span 256 byte page boundaries in the Flash memory space.

2.4 Instruction Formats

The M8C has a total of seven instruction formats which use instruction lengths of one, two, and three bytes. All instruction bytes are fetched from the program memory (Flash), using an address and data bus that are independent from the address and data buses used for register and RAM access.

While examples of instructions are given in this section, refer to the [M8C Instruction Set chapter on page 37](#) for detailed information on individual instructions.

2.4.1 One-Byte Instruction

Many instructions, such as some of the MOV instructions, have single-byte forms, because they do not use an address or data as an operand. As shown in [Table 2-4](#), one-byte instructions use an 8-bit opcode. The set of one-byte instructions can be divided into four categories, according to where their results are stored.

Table 2-4. One-Byte Instruction Format

Byte 0
8-Bit Opcode

The first category of one-byte instructions are those that do not update any registers or RAM. Only the one-byte no operation (NOP) and supervisory system call (SSC) instructions fit this category. While the program counter is incremented as these instructions execute, they do not cause any other internal M8C registers to be updated, nor do these instructions directly affect the register space or the RAM address space. The SSC instruction will cause SROM code to run, which will modify RAM and the M8C internal registers.

The second category has only the two PUSH instructions in it. The PUSH instructions are unique, because they are the only one-byte instructions that cause a RAM address to be modified. These instructions automatically increment the CPU_SP register (SP).

The third category has only the HALT instruction in it. The HALT instruction is unique, because it is the only one-byte instruction that causes a user register to be modified. The HALT instruction modifies user register space address FFh (CPU_SCR register).

The final category for one-byte instructions are those that cause updates of the internal M8C registers. This category holds the largest number of instructions: ASL, ASR, CPL, DEC, INC, MOV, POP, RET, RETI, RLC, ROMX, RRC, SWAP. These instructions can cause the CPU_A, CPU_X, and CPU_SP registers, or SRAM to update.

2.4.2 Two-Byte Instructions

The majority of M8C instructions are two bytes in length. While these instructions can be divided into categories identical to the one-byte instructions, this would not provide a useful distinction between the three two-byte instruction formats that the M8C uses.

Table 2-5. Two-Byte Instruction Formats

Byte 0		Byte 1	
4-Bit Opcode	12-Bit Relative Address		
8-Bit Opcode		8-Bit Data	
8-Bit Opcode		8-Bit Address	

The first two-byte instruction format, shown in the first row of [Table 2-5](#), is used by short jumps and calls: CALL, JMP, JACC, INDEX, JC, JNC, JNZ, JZ. This instruction format uses only four bits for the instruction opcode, leaving 12 bits to store the relative destination address in a two's-complement form. These instructions can change program execution to an address relative to the current address by -2048 or +2047.

The second two-byte instruction format, shown in the second row of [Table 2-5](#), is used by instructions that employ the Source Immediate addressing mode (see “[Source Immediate](#)” on [page 18](#)). The destination for these instructions is an internal M8C register, while the source is a constant value. An example of this type of instruction would be `ADD A, 7`.

The third two-byte instruction format, shown in the third row of [Table 2-5](#), is used by a wide range of instructions and addressing modes. The following is a list of the addressing modes that use this third two-byte instruction format:

- Source Direct (`ADD A, [7]`)
- Source Indexed (`ADD A, [X+7]`)
- Destination Direct (`ADD [7], A`)
- Destination Indexed (`ADD [X+7], A`)
- Source Indirect Post Increment (`MVI A, [7]`)
- Destination Indirect Post Increment (`MVI [7], A`)

For more information on addressing modes see “[Addressing Modes](#)” on [page 18](#).

2.4.3 Three-Byte Instructions

The three-byte instruction formats are the second most prevalent instruction formats. These instructions need three bytes because they either move data between two addresses in the user-accessible address space (registers and RAM) or they hold 16-bit absolute addresses as the destination of a long jump or long call.

Table 2-6. Three-Byte Instruction Formats

Byte 0	Byte 1	Byte 2
8-Bit Opcode	16-Bit Address (MSB, LSB)	
8-Bit Opcode	8-Bit Address	8-Bit Data
8-Bit Opcode	8-Bit Address	8-Bit Address

The first instruction format, shown in the first row of [Table 2-6](#), is used by the LJMP and LCALL instructions. These instructions change program execution unconditionally to an absolute address. The instructions use an 8-bit opcode, leaving room for a 16-bit destination address.

The second three-byte instruction format, shown in the second row of [Table 2-6](#), is used by the following two addressing modes:

- Destination Direct Source Immediate (`ADD [7], 5`)
- Destination Indexed Source Immediate (`ADD [X+7], 5`)

The third three-byte instruction format, shown in the third row of [Table 2-6](#), is for the Destination Direct Source Direct addressing mode, which is used by only one instruction. This instruction format uses an 8-bit opcode followed by two 8-bit addresses. The first address is the destination address in RAM, while the second address is the source address in RAM. The following is an example of this instruction:

```
MOV [7], [5]
```

2.5 Addressing Modes

The M8C has ten addressing modes:

- “Source Immediate” on page 18.
- “Source Direct” on page 19.
- “Source Indexed” on page 19.
- “Destination Direct” on page 20.
- “Destination Indexed” on page 20.
- “Destination Direct Source Immediate” on page 21.
- “Destination Indexed Source Immediate” on page 21.
- “Destination Direct Source Direct” on page 22.
- “Source Indirect Post Increment” on page 22.
- “Destination Indirect Post Increment” on page 23.

2.5.1 Source Immediate

For these instructions, the source value is stored in operand 1 of the instruction. The result of these instructions is placed in either the M8C CPU_A, CPU_F, or CPU_X register as indicated by the instruction’s opcode. All instructions using the Source Immediate addressing mode are two bytes in length.

Table 2-7. Source Immediate

Opcode	Operand 1
Instruction	Immediate Value

Source Immediate Examples:

Source Code	Machine Code	Comments
ADD A, 7	01 07	The immediate value 7 is added to the Accumulator. The result is placed in the Accumulator.
MOV X, 8	57 08	The immediate value 8 is moved into the CPU_X register.
AND F, 9	70 09	The immediate value of 9 is logically AND’ed with the CPU_F register and the result is placed in the CPU_F register.

2.5.2 Source Direct

For these instructions, the source address is stored in operand 1 of the instruction. During instruction execution, the address will be used to retrieve the source value from RAM or register address space. The result of these instructions is placed in either the M8C CPU_A or CPU_X register as indicated by the instruction's opcode. All instructions using the Source Direct addressing mode are two bytes in length.

Table 2-8. Source Direct

Opcode	Operand 1
Instruction	Source Address

Source Direct Examples:

Source Code	Machine Code	Comments
ADD A, [7]	02 07	The value in memory at address 7 is added to the Accumulator and the result is placed into the Accumulator.
MOV A, REG[8]	5D 08	The value in the register space at address 8 is moved into the Accumulator.

2.5.3 Source Indexed

For these instructions, the source offset from the CPU_X register is stored in operand 1 of the instruction. During instruction execution, the current CPU_X register value is added to the signed offset, to determine the address of the source value in RAM or register address space. The result of these instructions is placed in either the M8C CPU_A or CPU_X register as indicated by the instruction's opcode. All instructions using the Source Indexed addressing mode are two bytes in length.

Table 2-9. Source Indexed

Opcode	Operand 1
Instruction	Source Index

Source Indexed Examples:

Source Code	Machine Code	Comments
ADD A, [X+7]	03 07	The value in memory at address X+7 is added to the Accumulator. The result is placed in the Accumulator.
MOV X, [X+8]	59 08	The value in RAM at address X+8 is moved into the CPU_X register.

2.5.4 Destination Direct

For these instructions, the destination address is stored in the machine code of the instruction. The source for the operation is either the M8C CPU_A or CPU_X register as indicated by the instruction's opcode. All instructions using the Destination Direct addressing mode are two bytes in length.

Table 2-10. Destination Direct

Opcode	Operand 1
Instruction	Destination Address

Destination Direct Examples:

Source Code	Machine Code	Comments
ADD [7], A	04 07	The value in the Accumulator is added to memory at address 7. The result is placed in memory at address 7. The Accumulator is unchanged.
MOV REG[8], A	60 08	The Accumulator value is moved to register space at address 8. The Accumulator is unchanged.

2.5.5 Destination Indexed

For these instructions, the destination offset from the CPU_X register is stored in the machine code for the instruction. The source for the operation is either the M8C CPU_A register or an immediate value as indicated by the instruction's opcode. All instructions using the Destination Indexed addressing mode are two bytes in length.

Table 2-11. Destination Indexed

Opcode	Operand 1
Instruction	Destination Index

Destination Indexed Example:

Source Code	Machine Code	Comments
ADD [X+7], A	05 07	The value in memory at address X+7 is added to the Accumulator. The result is placed in memory at address X+7. The Accumulator is unchanged.

2.5.6 Destination Direct Source Immediate

For these instructions, the destination address is stored in operand 1 of the instruction. The source value is stored in operand 2 of the instruction. All instructions using the Destination Direct Source Immediate addressing mode are three bytes in length.

Table 2-12. Destination Direct Source Immediate

Opcode	Operand 1	Operand 2
Instruction	Destination Address	Immediate Value

Destination Direct Source Immediate Examples:

Source Code	Machine Code	Comments
ADD [7], 5	06 07 05	The value in memory at address 7 is added to the immediate value 5. The result is placed in memory at address 7.
MOV REG[8], 6	62 08 06	The immediate value 6 is moved into register space at address 8.

2.5.7 Destination Indexed Source Immediate

For these instructions, the destination offset from the CPU_X register is stored in operand 1 of the instruction. The source value is stored in operand 2 of the instruction. All instructions using the Destination Indexed Source Immediate addressing mode are three bytes in length.

Table 2-13. Destination Indexed Source Immediate

Opcode	Operand 1	Operand 2
Instruction	Destination Index	Immediate Value

Destination Indexed Source Immediate Examples:

Source Code	Machine Code	Comments
ADD [X+7], 5	07 07 05	The value in memory at address X+7 is added to the immediate value 5. The result is placed in memory at address X+7.
MOV REG[X+8], 6	63 08 06	The immediate value 6 is moved into the register space at address X+8.

2.5.8 Destination Direct Source Direct

Only one instruction uses this addressing mode. The destination address is stored in operand 1 of the instruction. The source address is stored in operand 2 of the instruction. The instruction using the Destination Direct Source Direct addressing mode is three bytes in length.

Table 2-14. Destination Direct Source Direct

Opcode	Operand 1	Operand 2
Instruction	Destination Address	Source Address

Destination Direct Source Direct Example:

Source Code	Machine Code	Comments
MOV [7], [8]	5F 07 08	The value in memory at address 8 is moved to memory at address 7.

2.5.9 Source Indirect Post Increment

Only one instruction uses this addressing mode. The source address stored in operand 1 is actually the address of a pointer. During instruction execution, the pointer's current value is read to determine the address in RAM where the source value is found. The pointer's value is incremented after the source value is read. For PSoC microcontrollers with more than 256 bytes of RAM, the Data Page Read (MVR_PP) register is used to determine which RAM page to use with the source address. Therefore, values from pages other than the current page can be retrieved without changing the Current Page Pointer (CUR_PP). The pointer is always read from the current RAM page. For information on the MVR_PP and CUR_PP registers, see the Register Reference chapter in the *PSoC Technical Reference Manual*. The instruction using the Source Indirect Post Increment addressing mode is two bytes in length.

Table 2-15. Source Indirect Post Increment

Opcode	Operand 1
Instruction	Source Address Pointer

Source Indirect Post Increment Example:

Source Code	Machine Code	Comments
MVI A, [8]	3E 08	The value in memory at address 8 (the indirect address) points to a memory location in RAM. The value at the memory location, pointed to by the indirect address, is moved into the Accumulator. The indirect address, at address 8 in memory, is then incremented.

2.5.10 Destination Indirect Post Increment

Only one instruction uses this addressing mode. The destination address stored in operand 1 is actually the address of a pointer. During instruction execution, the pointer's current value is read to determine the destination address in RAM where the Accumulator's value is stored. The pointer's value is incremented, after the value is written to the destination address. For PSoC microcontrollers with more than 256 bytes of RAM, the Data Page Write (MVW_PP) register is used to determine which RAM page to use with the destination address. Therefore, values can be stored in pages other than the current page without changing the Current Page Pointer (CUR_PP). The pointer is always read from the current RAM page. For information on the MVR_PP and CUR_PP registers, see the Register Reference chapter in the *PSoC Technical Reference Manual*. The instruction using the Destination Indirect Post Increment addressing mode is two bytes in length.

Table 2-16. Destination Indirect Post Increment

Opcode	Operand 1
Instruction	Destination Address Pointer

Destination Indirect Post Increment Example:

Source Code	Machine Code	Comments
MVI [8], A	3F 08	The value in memory at address 8 (the indirect address) points to a memory location in RAM. The Accumulator value is moved into the memory location pointed to by the indirect address. The indirect address, at address 8 in memory, is then incremented.

3. ImageCraft Assembler



This chapter details the information needed to use the ImageCraft Assembler. For information on command line usage of the ImageCraft Assembler, see the C Language Compiler User Guide (section 9.4 ImageCraft Assembler Arguments). For information on generating source code in PSoC Designer, see the PSoC Designer IDE Guide.

Assembly language is a low-level language. This means its structure is not like a human language. By comparison, 'C' is a high-level language with structures close to those used by human languages. Even though assembly is a low-level language, it is an abstraction created to make programming hardware easier for humans. Therefore, this abstraction must be eliminated before an input, in a form native to the microcontroller, can be generated. An assembler is used to convert the abstractions used in assembly language to machine code that the microcontroller can operate on directly.

3.1 Source File Format

Assembly language source files for the ImageCraft Assembler have five basic components as listed in [Table 3-1](#). Each line of the source file may hold a single label, mnemonic, comment, or directive. Multiple operands or expressions may be used on a single source file line. The maximum length for a line is 2,048 characters (including spaces) and the maximum word length is 256 characters. A word is a string of characters surrounded by spaces.

Table 3-1. Five Basic Components of an Assembly Source File

Component	Description
Label	Symbolic name followed by a colon (:).
Mnemonic	Character string representing an M8C instruction.
Operand	Arguments to M8C instructions.
Comment	May follow operands or expressions and starts in any column if first non-space character is either a C++-style comment (//) or semi-colon (;).
Directive	A command, interpreted by the Assembler, to control the generation of machine code.

Avoid use of the following characters in path and file names (they are problematic): \ / : * ? " < > | & + , ; = [] % \$ ` ' .

All user code is built from the components listed in [Table 3-1](#) and complex conditional-assembly constraints can be placed on a collection of source files. The text below has an example of each of the six basic components that will be discussed in detail in the following subsections. Line 1 is a comment line as indicated by the “//” character string. Lines 5, 6, and 7 also have comments starting with the “;” character and continuing to the end of the line. Lines 2 and 3 are examples of assembler directives. The character strings before the “:” character in lines 3 and 4 are labels. Lines 5, 6, and 7 have instruction mnemonics and operands.

```
Source File      1 // My Project Source Code
Components:     2 include "project.inc"
                3 BASE: equ      0x10
                4 _main:
                5 mov reg[0x00], 0x34    ;write 0x34 to Port 0
                6 mov A, reg[0x04]      ;read Port 1
                7 and [BASE+2], A       ;store Port 1 value in RAM
```

3.1.1 Labels

A label is a case-sensitive string of alphanumeric characters and underscores (_) followed by a colon. A label is assigned the address of the current Program Counter by the Assembler, unless the label is defined on a line with an EQU directive. See [“Equate Label EQU” on page 87](#) for more information. Labels can be placed on any line, including lines with source code as long as the label appears first. The Assembler supports three types of labels: local, global, and re-usable local.

Local Labels. These consist of a character string followed by a colon. Local labels cannot be referenced by other source files in the same project, they can only be used within the file in which they are defined. Local labels become global labels if they are “exported.” The following example has a single local label named SubFun. Local labels are case sensitive.

```
Local Labels:   mov X, 10

                SubFun:
                xor reg[00h], FFh
                dec X
                jnz SubFun
```

Global Labels. These are defined by the `EXPORT` assembler directive or by ending the label with two colons “:” rather than one. Global labels may be referenced from any source file in a project. The following example has two global labels. The `EXPORT` directive is used to make the `SubFun` label global, while two colons are used to make the `MoreFun` label global. Global labels are case sensitive.

```
Global Labels:      EXPORT SubFun
                    mov  X, 10

                    SubFun:
                    xor  reg[00h], FFh
                    dec  X
                    jnz  SubFun
                    mov  X, 5

                    MoreFun::
                    xor  reg[00h], FFh
                    dec  X
                    jnz  MoreFun
```

Re-usable Local Labels. These have multiple independent definitions within a single source file. They are defined by preceding the label string with a period “.”. The scope of a local label is bounded by the nearest local, or global label or the end of the source file. The following example has a single global label called `SubFun` and a re-usable local label called `.MoreFun`. Notice that while labels do not include the colon when referenced, re-usable local labels require that a period precede the label string for all instances. Re-usable local labels are case sensitive.

```
Re-usable Local    EXPORT SubFun
Label:              mov  X, 10

                    SubFun:
                    xor  reg[00h], FFh
                    mov  A, 5

                    .MoreFun:
                    xor  reg[04h], FFh
                    dec  A
                    jnz  .MoreFun
                    dec  X
                    jnz  SubFun
```

Local labels with the same name but in different source files is not supported.

3.1.2 Mnemonics

An instruction mnemonic is a two to five letter string that represents one of the microcontroller instructions. All mnemonics are defined in the [“Instruction Set Summary” on page 14](#). There can be 0 or 1 mnemonics per line of a source file. Mnemonics are not case sensitive.

3.1.3 Operands

Operands are the arguments to instructions. The number of operands and the format they use are defined by the instruction being used. The operand format for each instruction is covered in the [“Instruction Set Summary” on page 14](#).

Operands may take the form of constants, labels, dot operator, registers, RAM, or expressions.

Constants. These are operands bearing values explicitly stated in the source file. Constants may be stated in the source file using one of the radices listed in [Table 3-2](#).

Table 3-2. Constants Formats

Radix	Name	Formats	Example
127	ASCII Character	'J'	mov A, 'J' ;character constant mov A, '\\' ;use "\" to escape "\" mov A, '\\\\' ;use "\" to escape "\"
16	Hexadecimal	0x4A 4Ah \$4A	mov A, 0x4A ;hex--"0x" prefix mov A, 4Ah ;hex--append "h" mov A, \$4A ;hex--"\$" prefix
10	Decimal	74	mov A, 74 ;decimal--no prefix
8	Octal	0112	mov A, 0112 ;octal--zero prefix
2	Binary	0b01001010 %01001010	mov A, 0b01001010 ;bin--"0b" prefix mov A, %01001010 ;bin--"%" prefix

Labels. These may be used as an operand for an instruction, as described on [page 26](#). Labels are most often used as the operands for `jump` and `call` instructions to specify the destination address. However, labels may be used as an argument for any instruction.

Dot Operator (.). This is used to indicate that the ROM address of the first byte of the instruction should be used as an argument to the instruction.

```

Example 1:      mov A, <.      ; moves low byte of the PC to A
Example 2:      mov A, >.      ; moves high byte of the PC to A
Example 3:      jmp >.+3
                nop
                nop            ; jumped to this instruction
                nop

```

Registers. These have two forms in PSoC devices. The first type are those that exist in the two banks of user-accessible registers. The second type are those that exist in the microcontroller. [Table 3-3](#) contains examples for all types of register operands.

Table 3-3. Register Formats

Type	Formats	Example
User-Accessible Registers	reg[expr]	MOV A, reg[0x08] ;register at address 8 MOV A reg[OU+8] ;address = label OU + 8
M8C Registers	A	MOV A, 8 ;move 8 into the accumulator
	F	OR F, 1 ;set bit 0 of the flags
	SP	MOV SP, 8 ;set the stack pointer to 8
	X	MOV X, 8 ;set the M8C's X reg to 8

RAM. These references are made by enclosing the address or expression in square brackets. The Assembler will evaluate the expression to create the actual RAM address.

Table 3-4. RAM Format

Type	Formats	Example
Current RAM Page	[expr]	MOV A, [0x08] ;RAM at address 8 MOV A, [OU+8] ;address = label OU + 8

Expressions. These may be constructed using any combination of labels, constants, the dot operator, and the arithmetic and logical operations defined in [Table 3-5](#).

Table 3-5. Expressions

Precedence	Expression	Symbol	Form
1	Bitwise Complement	~	(~ a)
2	Multiplication	*	(a * b)
	Division	/	(a / b)
	Modulo	%	(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 re-locatable 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).

3.1.4 Comments

A comment starts with a semicolon (;) or a double slash (//) and goes 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 for reference.

3.1.5 Directives

An assembler directive is used to tell the ImageCraft Assembler to take some action during the assembly process. Directives are not understood by the M8C microcontroller. As such, directives allow the firmware writer to create code that is easier to maintain. See the [Assembler Directives chapter on page 75](#) for more information on directives.

3.2 Listing File Format

A `<project name>.lst` file is created each time the ImageCraft Assembler completes without errors or warnings. The list file may be used to understand how the Assembler has converted the source code into machine code.

The two lines below represent typical lines found in a listing file. Lines that begin with a four-digit number in parentheses (“()”) are source file lines. The number in parentheses is the source file line number. The text following the right parenthesis is the exact text from the source file. The second line in the example below begins with a four-digit number followed by a colon. This four-digit number indicates the ROM address for the first machine code byte that follows the colon. In this example, the two hexadecimal numbers that follow the colon are two bytes that form the `MOV A, 74` instruction. Notice that the ImageCraft Assembler converts the constants used in the source file to decimal values and that the machine code is always shown in hexadecimal. In this case the source code expressed the constant as an octal value (0112), the Assembler represented the same value in decimal (74), and the machine code uses hexadecimal (4A).

Example LST File: (0014) mov A, 0112 ; Octal constant
01AF: 50 4A MOV A,74

3.3 Map File Format

A `<project name>.mp` file is created each time the ImageCraft Assembler completes without errors or warnings.

The map file documents where the Assembler has placed areas defined by the `AREA` assembler directive and lists the values of global labels (also called global symbols).

The “`__data_start`” label shows the starting address of the RAM variables. InterruptRAM variables may overlap the “`__data_start`” label because “`__data_start`” is only used as a label by the compiler.

3.4 ROM File Format

A `<project name>.rom` file is created each time the ImageCraft Assembler completes without errors or warnings. This file is provided as an alternative to the Intel HEX file that is also created by the Assembler. The ROM file does not contain the user-defined protection settings for the Flash or the fill value used to initialize unused portions of Flash after the end of user code.

The ROM file is a simple text file with eight columns of data delimited by spaces. The example below is a complete ROM file for a 47-byte program. The ROM file does not contain any information about where the data should be located in Flash. By convention, the data in the ROM file starts at address

0x0000 in Flash. For the example below, only addresses 0x0000 through 0x002E of the Flash have assigned values according to the ROM file.

Example ROM	80 5B 00 00 7E 00 00 00
File:	7E 00 00 00 7D 02 62 7E
	7E 00 00 00 7D 01 EF 7E
	91 73 90 FE 90 89 90 14
	3D 7F 60 3A 5B 60 3E 7F
	3F 00 3D FF 3E CC FF

3.5 Intel® HEX File Format

The Intel HEX file created by the ImageCraft Assembler is used as a platform-independent way of distributing all of the information needed to program a PSoC microcontroller. In addition to the user data created by the Assembler, the HEX file also contains the protection settings for the project that will be used by the programmer.

The basic building block of the Intel HEX file format is called a record. Every record consists of six fields as shown in [Table 3-6](#). All fields, except for the start field, represent information as ASCII encoded hexadecimal. This means that every eight bits of information are encoded in two ASCII characters.

The start field is one byte in length and must always contain a colon (:). The length field is also one byte in length and indicates the number of bytes of data stored in the record. Because the length field is one byte in length, the maximum amount of data stored in a record is 255 bytes which would require 510 ASCII characters in the HEX file. The starting address field indicates the address of the first byte of information in the record. The address field is 16 bits in length (four ASCII characters) which allows room for 64 kilobytes of data per record.

Table 3-6. Intel HEX File Record Format

Field Number	Field Name	Length (bytes)	Description
1	start	1	The only valid value is the colon (:) character.
2	length	1	Indicates amount of data from 0 bytes to 255 bytes.
3	starting address	2	
4	type	1	"00": data "01": end of file "02": extended segment address "03": start segment address "04": extended linear address "05": start linear address record
5	data	Determined by length field	
6	checksum	1	

All HEX files created by the ImageCraft Assembler have the structure shown in [Table 3-7](#). Each row in the table describes a record type used in the HEX file. Each record type conforms to the record definitions discussed previously.

Table 3-7. PSoC Microcontroller Intel HEX File Format

Record	Description
<data record 1: flash data>	This is the first of many data records in the HEX file that contain Flash data.
<data record n: flash data>	The nth record containing data for Flash (last record). The total number of data records for Flash data can be determined by dividing the available Flash space (in bytes) by 64. Therefore, a 16 KB part would have a HEX file with 256 Flash data records.
:020000040010ea	The first two characters (02) indicate that this record has a length of two bytes (4 ASCII characters). The next four characters (0000) specify the starting address. The next two characters (04) indicate that this is an extended linear address. The four characters following 04 are the data for this record. Because this is an extended linear address record, the four characters indicate the value for the upper 16 bits of a 32-bit address. Therefore, the value of 0x0010 is a 1 MB offset. For PSoC microcontroller HEX files, the extended linear address is used to offset Flash protection data from the Flash data. The Flash protection bits start at the 1 MB address.
<data record 1: protection bits>	For PSoC devices with 16 KB of Flash or less, this is the only data record for protection bits.
<data record m: protection bits>	For PSoC devices with more than 16 KB of Flash, there will be an additional data record with protection bits for each 16 KB of additional Flash.
:020000040020da	This is another extended linear address record. This record provides a 1 MB offset from the Flash protection bits (absolute address of 2 MB).
<data record: checksum>	This is a two-byte data record that stores a checksum for all of the Flash data stored in the HEX file. The record will always start with :0200000000 and end with the four characters that represent the two-byte checksum.
:00000001ff	This is the end-of-file record. The length and starting address fields are all zero. The type field has a value of 0x01 and the checksum value will always be 0xff.

The following is an example of a PSoC device HEX file for a very small program.

```
Example Code:      mov    A, reg[0x04]
                   inc    A
                   mov    reg[0x04], A
```

Example ROM File: 5D 04 74 60 04

[illegible]

Records removed to make example compact.

[illegible]

3.6 Convention for Restoring Internal Registers

When calling PSoC user module APIs and library functions, it is the caller's responsibility to preserve the CPU_A and CPU_X registers. This means that if the current context of the code has a value in the CPU_X and/or CPU_A register that must be maintained after the API call, then the caller must save (push on the stack) and then restore (pop off the stack) them after the call has returned.

Even though some of the APIs do preserve the CPU_X and CPU_A register, Cypress reserves the right to modify the API in future releases in such a manner as to modify the contents of the CPU_X and CPU_A registers. Therefore, it is very important to observe the convention when calling from assembly. Note that the C compiler observes this convention.

3.7 Compiling a File into a Library Module

Each library module is simply an object file. Therefore, to create a library module, you need to compile a source file into an object file. There are several ways that you can create a library.

One method is to create a brand new project. Add all the necessary source files that you wish to be added to your custom library to this project. You then add a project-specific MAKE file action to add those project files to a custom library.

For example, a blank project is created for any type of part, since interest is only in using 'C' and/or assembly, the Application Editor, and the Debugger. The goal for creating a custom library is to centralize a set of common functions that can be shared between projects. These common functions, or primitives, have deterministic inputs and outputs. Another goal for creating this custom library is to be able to debug the primitives using a sequence of test instructions (e.g., a regression test) in a source file that should not be included in the library. No user modules are involved in this example.

PSoC Designer automatically generates a certain amount of code for each new project. In this example, use the generated `_main` source file to hold regression tests, but do not add this file to the custom library. Also, do not add the generated `boot.asm` source file to the library. Essentially, all the files under the "Source Files" branch of the project view source tree go into a custom library, except `main.asm` (or `main.c`) and `boot.asm`.

Create a file called `local.dep` in the root folder of the project. The `local.dep` file is included by the master `Makefile` (found in the `...\PSoC Designer\tools` folder). The following shows how the `Makefile` includes `local.dep` (found at the bottom of `Makefile`).

```
#this include is the dependencies
-include project.dep
#if you like project.dep that is good!
-include local.dep
```

The nice thing about having `local.dep` included at the end of the master `Makefile` is that the rules used in the `Makefile` can be redefined (see the Help > Documentation \Supporting Documents\make.pdf for detailed information). In this example, it is used as an advantage.

The following shows information from example `local.dep`.

```
# ----- Cut/Paste to your local.dep File -----
define Add_To_MyCustomLib
$(CRLF)
$(LIBCMD) -a PSoCToolsLib.a $(library_file)
endif
obj/%.o : %.asm project.mk
ifeq ($(ECHO_COMMANDS),novice)
    echo $(call correct_path,$<)
endif
    $(ASMCMD) $(INCLUDEFLAGS) $(DEFAULTASMFLAGS) $(ASMFLAGS) -    $@ $(call
correct_path,$<)
    $(foreach library_file, $(filter-out obj/main.o, $@),
    $(Add_To_MyCustomLib))
obj/%.o : %.c project.mk
ifeq ($(ECHO_COMMANDS),novice)
    echo $(call correct_path,$<)
endif
    $(CCMD) $(CFLAGS) $(CDEFINES) $(INCLUDEFLAGS)
    $(DEFAULTCFLAGS) -o $@ $(call correct_path,$<)
    $(foreach library_file, $(filter-out obj/main.o, $@),
    $(Add_To_MyCustomLib))
# ----- End Cut -----
```

The rules (for example, `obj/%.o : %.asm project.mk` and `obj/%.o : %.c project.mk`) in the *local.dep* file shown above are the same rules found in the master *Makefile* with one addition each. The addition in the redefined rules is to add each object (target) to a library called *PSoCToolsLib.a*. For example:

```
$(foreach library_file, $(filter-out obj/main.o,
$@), $(Add_To_MyCustomLib))
```

The MAKE keyword `foreach` causes one piece of text (the first argument) to be used repeatedly, each time with a different substitution performed on it. The substitution list comes from the second `foreach` argument.

In this second argument, there is another MAKE keyword/function called `filter-out`. The `filter-out` function removes `obj/main.o` from the list of all targets being built (for example, `obj/%.o`). This was one of the goals for this example. You can filter out additional files by adding those files to the first argument of `filter-out` such as:

```
$(filter-out obj/main.o obj/excludeme.o, $@).
```

The MAKE symbol combination `$@` is a shortcut syntax that refers to the list of all the targets (for example, `obj/%.o`).

The third argument in the `foreach` function is expanded into a sequence of commands, for each substitution, to update or add the object file to the library. This *local.dep* example is prepared to handle both C and assembly source files and put them in the library, *PSoCToolsLib.a*. The library is created/updated in the project root folder in this example. However, you can provide a full path to another folder. For example:

```
$(LIBCMD) -a c:\temp\PSoCToolsLib.a $(library_file.
```

Another goal was to not include the *boot.asm* file in the library. This is easy given that the master *Makefile* contains a separate rule for the *boot.asm* source file, which is not redefined in *local.dep*.

You can cut and paste this example and place it in a *local.dep* file in the root folder of any project. To view messages in the Build tab of the Output Status window regarding the behavior of your custom process, go to Tools > Options > Builder tab and click a check at "Use verbose build messages."

Use the Project > Settings > Linker tab fields to add the library modules/library path if you want other PSoC Designer projects to link in your custom library.

4. M8C Instruction Set



This chapter describes all M8C instructions in detail. The M8C supports a total of 256 instructions which are divided into 37 instruction types and arranged in alphabetical order according to the instruction types mnemonic.

For each instruction the assembly code format will be illustrated as well as the operation performed by the instruction. The microprocessor cycles that are listed for each instruction are for instructions that are not on a ROM (Flash) page-boundary execution. If the instruction is located on a 256-byte ROM page boundary, an additional microprocessor clock cycle will be needed by the instruction. The `expr` string that is used to explain the assembly code format represents the use of assembler directives which tell the ImageCraft Assembler how to calculate the constant used in the final machine code. Note that in the operation equations the machine code constant is represented by k , k_1 , and k_2 .

While the instruction mnemonics are often shown in all capital letters, the ImageCraft Assembler ignores case for directives and instructions mnemonics. However, the Assembler does consider case for user-defined symbols (i.e., labels).

Note that information about individual M8C instructions is also available via *PSoC Designer Online Help*. Pressing the [F1] key will cause the online help system to search for the word at the current insertion point in a source file. If your insertion point is an instruction mnemonic, pressing [F1] will direct you to information about that instruction.

4.1 Add with Carry

ADC

Computes the sum of the two operands plus the carry value from the Flag register. The first operand's value is replaced by the computed sum. If the sum is greater than 255, the Carry Flag is set in the Flag register. If the sum is zero, the Zero Flag is set in the Flag register.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
ADC	A, expr	$A \leftarrow A + k + CF$	0x09	4	2
ADC	A, [expr]	$A \leftarrow A + \text{ram}[k] + CF$	0x0A	6	2
ADC	A, [X+expr]	$A \leftarrow A + \text{ram}[X + k] + CF$	0x0B	7	2
ADC	[expr], A	$\text{ram}[k] \leftarrow \text{ram}[k] + A + CF$	0x0C	7	2
ADC	[X+expr], A	$\text{ram}[X + k] \leftarrow \text{ram}[X + k] + A + CF$	0x0D	8	2
ADC	[expr], expr	$\text{ram}[k_1] \leftarrow \text{ram}[k_1] + k_2 + CF$	0x0E	9	3
ADC	[X+expr], expr	$\text{ram}[X + k_1] \leftarrow \text{ram}[X + k_1] + k_2 + CF$	0x0F	10	3

Conditional Flags: CF Set if the sum > 255; cleared otherwise.
ZF Set if the result is zero; cleared otherwise.

Example 1:

```

mov  A, 0           ;set accumulator to zero
or   F, 0x02        ;set carry flag
adc  A, 12           ;accumulator value is now 13

```

Example 2:

```

mov  [0x39], 0       ;initialize ram[0x39]=0x00
mov  [0x40], FFh     ;initialize ram[0x40]=0xFF
inc  [0x40]          ;ram[0x40]=0x00, CF=1, ZF=1
adc  [0x39], 0       ;ram[0x39]=0x01, CF=0, ZF=0

```

4.2 Add without Carry

ADD

Computes the sum of the two operands. The first operand's value is replaced by the computed sum. If the sum is greater than 255, the Carry Flag is set in the Flag register. If the sum is zero, the Zero Flag is set in the Flag register. The `ADD SP, expr` instruction does not affect the flags in any way.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
ADD	A, expr	$A \leftarrow A + k$	0x01	4	2
ADD	A, [expr]	$A \leftarrow A + \text{ram}[k]$	0x02	6	2
ADD	A, [X+expr]	$A \leftarrow A + \text{ram}[X + k]$	0x03	7	2
ADD	[expr], A	$\text{ram}[k] \leftarrow \text{ram}[k] + A$	0x04	7	2
ADD	[X+expr], A	$\text{ram}[X + k] \leftarrow \text{ram}[X + k] + A$	0x05	8	2
ADD	[expr], expr	$\text{ram}[k_1] \leftarrow \text{ram}[k_1] + k_2$	0x06	9	3
ADD	[X+expr], expr	$\text{ram}[X + k_1] \leftarrow \text{ram}[X + k_1] + k_2$	0x07	10	3
ADD	SP, expr	$SP \leftarrow SP + k$	0x38	5	2

Conditional Flags:

CF Set if the sum > 255; cleared otherwise.
`ADD SP, expr` does not affect the Carry Flag.

ZF Set if the result is zero; cleared otherwise.
`ADD SP, expr` does not affect the Zero Flag.

Example 1:

```
mov A, 10      ;initialize A to 10 (decimal)
add A, 240     ;result is A=250 (decimal)
add A, 6       ;result is A=0, CF=1, ZF=1
```

Example 2:

```
mov A, 10      ;initialize A to 10 (decimal)
add A, 240     ;result is A=250 (decimal)
add A, 7       ;result is A=1, CF=1, ZF=0
add A, 5       ;result is A=6, CF=0, ZF=0
```

Example 3:

```
mov A, 10      ;initialize A to 10 (decimal)
swap A, SP     ;put 10 in SP
add SP, 240    ;result is SP=250 (decimal)
add SP, 6      ;SP=0, CF=unchanged, ZF=unchanged
```

4.3 Bitwise AND

AND

Computes the logical AND for each bit position using both arguments. The result of the logical AND is placed in the corresponding bit position for the first argument.

The Carry Flag is only changed when the `AND F, expr` instruction is used. The CF will be set to the result of the logical AND of the CF at the beginning of instruction execution and the second argument's value at bit position 2 (i.e., `F[2]` and `expr[2]`).

For the `AND F, expr` instruction the ZF is handled the same as the CF in that it is changed as a result of the logical AND of the ZF's value at the beginning of instruction execution and the value of the second argument's value at bit position 1 (i.e., `F[1]` and `expr[1]`). However, for all other `AND` instructions the Zero Flag will be set or cleared based on the result of the logical AND operation. If the result of the AND is that all bits are zero, the Zero Flag will be set; otherwise, the Zero Flag is cleared.

Note that AND (or OR or XOR, as appropriate) is a read-modify write instruction. When operating on a register, that register must be of the read-write type. Bitwise AND to a write only register will generate nonsense.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
AND	A, expr	$A \leftarrow A \& k$	0x21	4	2
AND	A, [expr]	$A \leftarrow A \& \text{ram}[k]$	0x22	6	2
AND	A, [X+expr]	$A \leftarrow A \& \text{ram}[X+k]$	0x23	7	2
AND	[expr], A	$\text{ram}[k] \leftarrow \text{ram}[k] \& A$	0x24	7	2
AND	[X+expr], A	$\text{ram}[X+k] \leftarrow \text{ram}[X+k] \& A$	0x25	8	2
AND	[expr], expr	$\text{ram}[k_1] \leftarrow \text{ram}[k_1] \& k_2$	0x26	9	3
AND	[X+expr], expr	$\text{ram}[X+k_1] \leftarrow \text{ram}[X+k_1] \& k_2$	0x27	10	3
AND	REG[expr], expr	$\text{reg}[k_1] \leftarrow \text{reg}[k_1] \& k_2$	0x41	9	3
AND	REG[X+expr], expr	$\text{reg}[X+k_1] \leftarrow \text{reg}[X+k_1] \& k_2$	0x42	10	3
AND	F, expr	$F \leftarrow F \& k$	0x70	4	2

Conditional Flags: CF Affected only by the `AND F, expr` instruction.
 ZF Set if the result is zero; cleared otherwise.
`AND F, expr` will set this flag as a result of the AND operation.

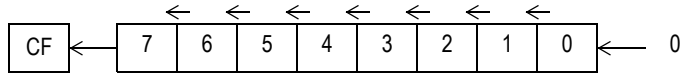
Example 1: `and A, 0x00 ; A=0, CF=unchanged, ZF=1`

Example 2: `and F, 0x00 ; F=0 therefore CF=0, ZF=0`

4.4 Arithmetic Shift Left

ASL

Shifts all bits of the instruction's argument one bit to the left. Bit 7 is loaded into the Carry Flag and bit 0 is loaded with a zero.



Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
ASL	A	$\begin{aligned} & \text{CF} \leftarrow \text{A}:7 \\ & \text{A}:7 \leftarrow \text{A}:6 \\ & \text{A}:6 \leftarrow \text{A}:5 \\ & \text{A}:5 \leftarrow \text{A}:4 \\ & \text{A}:4 \leftarrow \text{A}:3 \\ & \text{A}:3 \leftarrow \text{A}:2 \\ & \text{A}:2 \leftarrow \text{A}:1 \\ & \text{A}:1 \leftarrow \text{A}:0 \\ & \text{A}:0 \leftarrow 0 \end{aligned}$	0x64	4	1
ASL	[expr]	$\begin{aligned} & \text{CF} \leftarrow \text{ram}[k]:7 \\ & \text{ram}[k]:7 \leftarrow \text{ram}[k]:6 \\ & \text{ram}[k]:6 \leftarrow \text{ram}[k]:5 \\ & \text{ram}[k]:5 \leftarrow \text{ram}[k]:4 \\ & \text{ram}[k]:4 \leftarrow \text{ram}[k]:3 \\ & \text{ram}[k]:3 \leftarrow \text{ram}[k]:2 \\ & \text{ram}[k]:2 \leftarrow \text{ram}[k]:1 \\ & \text{ram}[k]:1 \leftarrow \text{ram}[k]:0 \\ & \text{ram}[k]:0 \leftarrow 0 \end{aligned}$	0x65	7	2
ASL	[X+expr]	$\begin{aligned} & \text{CF} \leftarrow \text{ram}[(X+k)]:7 \\ & \text{ram}[(X+k)]:7 \leftarrow \text{ram}[(X+k)]:6 \\ & \text{ram}[(X+k)]:6 \leftarrow \text{ram}[(X+k)]:5 \\ & \text{ram}[(X+k)]:5 \leftarrow \text{ram}[(X+k)]:4 \\ & \text{ram}[(X+k)]:4 \leftarrow \text{ram}[(X+k)]:3 \\ & \text{ram}[(X+k)]:3 \leftarrow \text{ram}[(X+k)]:2 \\ & \text{ram}[(X+k)]:2 \leftarrow \text{ram}[(X+k)]:1 \\ & \text{ram}[(X+k)]:1 \leftarrow \text{ram}[(X+k)]:0 \\ & \text{ram}[(X+k)]:0 \leftarrow 0 \end{aligned}$	0x66	8	2

Conditional Flags: CF Set equal to the initial argument's bit 7 value.
ZF Set if the result is zero; cleared otherwise.

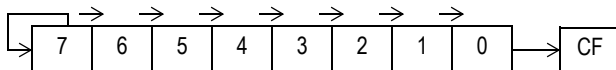
Example 1: `mov A, 0x7F ; initialize A with 127`
`asl A ; A=0xFE, CF=0, ZF=0`

Example 2: `mov 0xEB], AA ; initialize RAM @ 0xEB with 0`
`asl 0xEB] ; ram[0xEB]=54, CF=1, ZF=0`

4.5 Arithmetic Shift Right

ASR

Shifts all bits of the instruction's argument one bit to the right. Bit 7 remains the same while bit 0 is shifted into the Carry Flag.



Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
ASR	A	$A \leftarrow \begin{bmatrix} CF \leftarrow A:0, A:0 \leftarrow A:1, A:1 \leftarrow A:2 \\ A:2 \leftarrow A:3, A:3 \leftarrow A:4, A:4 \leftarrow A:5 \\ A:5 \leftarrow A:6, A:6 \leftarrow A:7 \end{bmatrix}$	0x67	4	1
ASR	[expr]	$ram[k] \leftarrow \begin{bmatrix} CF \leftarrow ram[k]:0 \\ ram[k]:0 \leftarrow ram[k]:1 \\ ram[k]:1 \leftarrow ram[k]:2 \\ ram[k]:2 \leftarrow ram[k]:3 \\ ram[k]:3 \leftarrow ram[k]:4 \\ ram[k]:4 \leftarrow ram[k]:5 \\ ram[k]:5 \leftarrow ram[k]:6 \\ ram[k]:6 \leftarrow ram[k]:7 \end{bmatrix}$	0x68	7	2
ASR	[X+expr]	$ram[X+k] \leftarrow \begin{bmatrix} CF \leftarrow ram[(X+k)]:0 \\ ram[(X+k)]:0 \leftarrow ram[(X+k)]:1 \\ ram[(X+k)]:1 \leftarrow ram[(X+k)]:2 \\ ram[(X+k)]:2 \leftarrow ram[(X+k)]:3 \\ ram[(X+k)]:3 \leftarrow ram[(X+k)]:4 \\ ram[(X+k)]:4 \leftarrow ram[(X+k)]:5 \\ ram[(X+k)]:5 \leftarrow ram[(X+k)]:6 \\ ram[(X+k)]:6 \leftarrow ram[(X+k)]:7 \end{bmatrix}$	0x69	8	2

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

Example 1: `mov A, 0x00 ;initialize A to 0`
`and F, 0x00 ;make sure all flags are cleared`
`asr A ;A=0, CF=0, ZF=1`

Example 2: `mov A, 0xFF ;initialize A to 255`
`and F, 0x00 ;make sure all flags are cleared`
`asr A ;A=0xFF, CF=1, ZF=0`

Example 3: `mov A, 0xAA ;initialize A to 170`
`and F, 0x00 ;make sure all flags are cleared`
`asr A ;A=0xD5, CF=0, ZF=0`

4.6 Call Function

CALL

Adds the signed argument to the current PC+2 value resulting in a new PC that determines the address of the first byte of the next instruction. The current PC value is defined as the PC value that corresponds to the ROM address of the first byte of the next instruction.

Two pushes are used to store the Program Counter (PC+2) on the stack. First, the upper 8 bits of the PC (CPU_PC register) are placed on the stack followed by the lower 8 bits. The Stack Pointer is post-incremented for each push. For devices with more than 256 bytes of RAM, the stack is confined to a single designated stack page defined in the device data sheet. The M8C automatically selects the stack page as the destination for the push during the `CALL` instruction. Therefore, a `CALL` instruction may be issued in any RAM page. After the `CALL` instruction has completed, user code will be operating from the same RAM page as before the `CALL` instruction was executed.

This instruction has a 12-bit two's-complement relative address that is added to the PC. The 12 bits are packed into the two-byte instruction format by using the lower nibble of the opcode and the second byte of the instruction format. Therefore, all opcodes with an upper nibble of 9 are `CALL` instructions. The "x" character is used in the table below to indicate that the first byte of a `CALL` instruction can have one of 16 values (i.e., 0x90, 0x91, 0x92,..., 0x9F).

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
CALL	expr	$PC \leftarrow PC + 2 + k, (-2048 \leq k \leq 2047)$	0x9x	11	2

Conditional Flags: CF Unaffected.
ZF Unaffected.

Example:

```

0000          _main:
0000 40        nop
0001 90 E8     call SubFun
0003 40        nop

```

Note that the relative address for the `CALL` above is positive (0xE8) and that the sum of that address and the PC value for the first byte of the next instruction (0x0003) equals the address of the `SubFun` label (0xE8 + 0x0003 = 0x00EB).

```

0004 9F FA     call _main

```

Note that the call to `Main` uses a negative address (0xFA).

```

0006
00EB          org 0x00EB
00EB          SubFun:
00EB 40        nop
00EC 7F        ret

```

4.7 Non-Destructive Compare

CMP

Subtracts the second argument from the first. If the difference is less than zero the Carry Flag is set. If the difference is 0 the Zero Flag is set. Neither operand's value is destroyed by this instruction.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
CMP	A, expr	$A - k$	0x39	5	2
CMP	A, [expr]	$A - \text{ram}[k]$	0x3A	7	2
CMP	A, [X+expr]	$A - \text{ram}[X + k]$	0x3B	8	2
CMP	[expr] ₁ , expr ₂	$\text{ram}[k_1] - k_2$	0x3C	8	3
CMP	[X+expr] ₁ , expr ₂	$\text{ram}[X + k_1] - k_2$	0x3D	9	3

Conditional Flags: CF Set if Operand 1 < Operand 2; cleared otherwise.
 ZF Set if the operands are equal; cleared otherwise.

Example:

```

mov  A, 34      ;initialize the accumulator to 34
cmp  A, 33      ;A>=34 CF cleared, A != 33 ZF cleared
cmp  A, 34      ;A=34 CF cleared, ZF set
cmp  A, 35      ;A<35 CF set, A != 35 ZF cleared

```

4.8 Complement Accumulator

CPL

Computes the bitwise complement of the Accumulator and stores the result in the Accumulator. The Carry Flag is not affected but the Zero Flag will be set, if the result of the complement is '0' (for example, the original value was 0xFF).

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
CPL	A	$A \leftarrow \bar{A}$	0x73	4	1

Conditional Flags: CF Unaffected.
ZF Set if the result is zero; cleared otherwise.

Example 1: `mov A, 0xFF`
`cpl A` ; A=0x00, ZF=1

Example 2: `mov A, 0xA5`
`cpl A` ; A=0x5A, ZF=0

Example 3: `mov A, 0xFE`
`cpl A` ; A=0x01, ZF=0

4.9 Decrement

DEC

Subtracts one from the value of the argument and replaces the argument's original value with the result. If the result is '-1' (original value was zero) the Carry Flag is set. If the result is '0' (original value was one) the Zero Flag is set.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
DEC	A	$A \leftarrow A - 1$	0x78	4	1
DEC	X	$X \leftarrow X - 1$	0x79	4	1
DEC	[expr]	$\text{ram}[k] \leftarrow \text{ram}[k] - 1$	0x7A	7	2
DEC	[X+expr]	$\text{ram}[X + k] \leftarrow \text{ram}[X + k] - 1$	0x7B	8	2

Conditional Flags: CF Set if the result is -1; cleared otherwise.
ZF Set if the result is zero; cleared otherwise.

Example:

```

mov  [0xEB], 3
loop2:                ;The loop will be executed 3 times.
dec  [0xEB]
jnz  loop2            ;Jump will not be taken when ZF is set by
                        ;DEC (i.e., wait until the loop counter
                        ;(0xEB) is decremented to 0x00).
```



HALT

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
HALT		$\text{reg}[\text{CPU_SCR}] \leftarrow \text{reg}[\text{CPU_SCR}] + 1$	0x30	9	1

Example: halt ;sets STOP bit in CPU SCR register

4.11 Increment

INC

Adds one to the argument. The argument's original value is replaced by the new value. If the value after the increment is 0x00, the Carry Flag and the Zero Flag will be set (original value must have been 0xFF).

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
INC	A	$A \leftarrow A + 1$	0x74	4	1
INC	X	$X \leftarrow X + 1$	0x75	4	1
INC	[expr]	$\text{ram}[k] \leftarrow \text{ram}[k] + 1$	0x76	7	2
INC	[X+expr]	$\text{ram}[X+k] \leftarrow \text{ram}[X+k] + 1$	0x77	8	2

Conditional Flags: CF Set if value after the increment is 0; cleared otherwise.
ZF Set if the result is zero; cleared otherwise.

Example 1:

```

mov  A, 0x00      ;initialize A to 0
or   F, 0x06      ;make sure CF and ZF are set (1)
inc  A            ;A=0x01, CF=0, ZF=0

```

Example 2:

```

mov  A, 0xFF      ;initialize A to 0
and  F, 0x00      ;make sure flags are all 0
inc  A            ;A=0x00, CF=1, ZF=1

```


4.12 Relative Table Read

INDEX

Places the contents of ROM at the location indicated by the sum of the Accumulator, the argument, and the current PC+2 into the Accumulator. This instruction has a 12-bit, two's-complement offset address, relative to the current PC+2. The current PC value is defined as the PC value that corresponds to the ROM address of the first byte of the instruction.

The `INDEX` instruction is used to retrieve information from a table to the Accumulator. The lower nibble of the first byte of the instruction is used as the upper 4 bits of the 12-bit address. Therefore, all instructions that begin with 0xF are `INDEX` instructions, so all of the following are `INDEX` opcodes: 0xF0, 0xF1, 0xF2,..., 0xFF.

The offset into the table is taken as the value of the Accumulator when the `INDEX` instruction is executed. The maximum readable table size is 256 bytes due to the Accumulator being 8 bits in length.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
INDEX	expr	$A \leftarrow \text{rom}[k + A + PC + 2], (-2048 \leq k \leq 2047)$	0xFx	13	2

Conditional Flags: CF Unaffected.
ZF Set if the byte returned to A is zero.

Example:

```

0000                                OUT_REG: equ 04h
0000 40                            [04]  nop
0001 50 03                         [04]  mov A, 3
0003 F0 E6                         [13]  index ASCIINumbers
0005 60 04                         [05]  mov reg[OUT_REG], A

```

Note that the 12-bit address for the `INDEX` instruction is positive and that the sum of the address (0x0E6) and the next instruction's address (0x0005) are equal to the first address of the ASCIINumbers table (0x00EB). Because the Accumulator has been set to 3 before executing the `INDEX` instruction, the fourth byte in the ASCIINumbers table will be returned to A. Therefore, A will be 0x33 at the end of the `INDEX` instruction.

```

0007
00EB                                org 0x00EB
00EB                                ASCIINumbers:
00EB 30 31 ...                      ds      "0123456789"
                                32 33 34 35 36 37 38 39

```

4.13 Jump Accumulator

JACC

Jump, unconditionally, to the address computed by the sum of the Accumulator, the 12-bit two's-complement argument, and the current PC+1. The current PC value is defined as the PC value that corresponds to the ROM address of the first byte of the `JACC` instruction.

The Accumulator is not affected by this instruction. The `JACC` instruction uses a two-byte instruction format where the lower nibble of the first byte is used for the upper 4 bits of the 12-bit relative address. This causes an effective 4-bit opcode. Therefore, the following are all valid opcode bytes for the `JACC` instruction: 0xE0, 0xE1, 0xE2,..., 0xEF.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
JACC	expr	$PC \leftarrow (PC + 1) + k + A$	0xE _x	7	2

Conditional Flags: CF Unaffected.
 ZF Unaffected.

Example: 0000 `_main:`
 0000 50 03 `mov A, 3` ;set A with jump offset
 0002 E0 01 `jacc SubFun`

Program execution will jump to address 0x0007 (halt)

```

0004 SubFun:
0004 40      nop
0005 40      nop
0006 40      nop
0007 30      halt

```

4.14 Jump if Carry

JC

If the Carry Flag is set, jump to the sum of the relative address argument and the current PC+1. The current PC value is defined as the PC value that corresponds to the ROM address of the first byte of the JC instruction.

The JC instruction uses a two-byte instruction format where the lower nibble of the first byte is used for the upper 4 bits of the 12-bit relative address. This causes an effective 4-bit opcode. Therefore, the following are all valid opcode bytes for the JC instruction: 0xC0, 0xC1, 0xC2,..., 0xCF.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
JC	expr	$PC \leftarrow (PC + 1) + k, (-2048 \leq k \leq 2047)$	0xCx	5	2

Conditional Flags: CF Unaffected.
ZF Unaffected.

```

Example:  0000          _main:
          0000  55 3C 02  mov [3Ch], 2
          0003  16 3C 03  sub [3Ch], 3      ;2-2=0 CF=1, ZF=0
          0006  C0 02    jc SubFun          ;CF=1, jump to SubFun
          0008  30      halt
          0009
          0009          SubFun:
          0009  40      nop

```

4.15 Jump

JMP

Jump, unconditionally, to the address indicated by the sum of the argument and the current PC+1. The current PC value is defined as the PC value that corresponds to the ROM address of the first byte of the `JMP` instruction.

The `JMP` instruction uses a two-byte instruction format where the lower nibble of the first byte is used for the upper 4 bits of the 12-bit relative address. This causes an effective 4-bit opcode. Therefore, the following are all valid opcode bytes for the `JMP` instruction: 0x80, 0x81, 0x82,..., 0x8F.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
JMP	expr	$PC \leftarrow (PC + 1) + k, (-2048 \leq k \leq 2047)$	0x8x	5	2

Conditional Flags: CF Unaffected.
ZF Unaffected.

Example: 0000 _main:
0000 80 01 [05] jmp SubFun
Jump is forward, relative to PC, therefore offset is positive (0x01).

0002 SubFun:
0002 8F FD [05] jmp _main
Jump is backwards, relative to PC, therefore, offset is negative (0xFD).

4.16 Jump if No Carry

JNC

If the Carry Flag is not set, jump to the sum of the relative address argument and the current PC+1. The current PC value is defined as the PC value that corresponds to the ROM address of the first byte of the JNC instruction.

The JNC instruction uses a two-byte instruction format where the lower nibble of the first byte is used for the upper 4 bits of the 12-bit relative address. This causes an effective 4-bit opcode. Therefore, the following are all valid opcode bytes for the JNC instruction: 0xD0, 0xD1, 0xD2,..., 0xDF.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
JNC	expr	$PC \leftarrow (PC + 1) + k, (-2048 \leq k \leq 2047)$	0xDx	5	2

Conditional Flags: CF Unaffected.
ZF Unaffected.

```

Example:  0000                                _main:
           0000 55 3C 02  [08]  mov [3Ch], 2
           0003 16 3C 02  [09]  sub [3Ch], 2    ;2-2=0 CF=0, ZF=1
           0006 D0 02    [05]  jnc SubFun      ;jump to SubFun
           0008 30      [04]  halt
           0009
           0009                                SubFun:
           0009 40      [04]  nop

```

4.17 Jump if Not Zero

JNZ

If the Zero Flag is not set, jump to the address indicated by the sum of the argument and the current PC+1. The current PC value is defined as the PC value that corresponds to the ROM address of the first byte of the JNZ instruction.

The JNZ instruction uses a two-byte instruction format where the lower nibble of the first byte is used for the upper 4 bits of the 12-bit relative address. This causes an effective 4-bit opcode. Therefore, the following are all valid opcode bytes for the JNZ instruction: 0xB0, 0xB1, 0xB2,..., 0xBF.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
JNZ	expr	$PC \leftarrow (PC + 1) + k, (-2048 \leq k \leq 2047)$	0xBx	5	2

Conditional Flags: CF Unaffected.
ZF Unaffected.

```

Example:  0000          _main:
           0000 55 3C 02  [08]  mov [3Ch], 2
           0003 16 3C 01  [09]  sub [3Ch], 1    ;2-1=1 CF=0, ZF=0
           0006 B0 02     [05]  jnz SubFun      ;jump to SubFun
           0008 30        [04]  halt
           0009
           0009          SubFun:
           0009 40        [04]  nop

```

4.18 Jump if Zero

JZ

If the Zero Flag is set, jump to the address indicated by the sum of the argument and the current PC+1. The current PC value is defined as the PC value that corresponds to the ROM address of the first byte of the JZ instruction.

The JZ instruction uses a two-byte instruction format where the lower nibble of the first byte is used for the upper 4 bits of the 12-bit relative address. This causes an effective 4-bit opcode. Therefore, the following are all valid opcode bytes for the JZ instruction: 0xA0, 0xA1, 0xA2,..., 0xAF.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
JZ	expr	$PC \leftarrow (PC + 1) + k, (-2048 \leq k \leq 2047)$	0xAx	5	2

Conditional Flags: CF Unaffected.
ZF Unaffected.

```

Example:  0000          _main:
          0000 55 3C 02 [08]  mov [3Ch], 2
          0003 16 3C 02 [09]  sub [3Ch], 2      ;2-2=0 CF=0, ZF=1
          0006 A0 02   [05]  jz SubFun          ;jump to SubFun
          0008 30     [04]  halt
          0009
          0009          SubFun:
          0009 40     [04]  nop

```

4.19 Long Call

LCALL

Replaces the PC value with the `LCALL` instruction's argument. The new PC value determines the address of the first byte of the next instruction.

Two pushes are used to store the Program Counter (current PC+3) on the stack. The current PC value is defined as the PC value that corresponds to the ROM address of the first byte of the instruction.

First, the upper 8 bits of the PC+3 are placed on the stack followed by the lower 8 bits. The Stack Pointer is post-incremented for each push. For PSoC microcontrollers with more than 256 bytes of RAM, the stack is confined to a single designated stack page defined in the device data sheet. The M8C automatically selects the stack page as the destination for the push during the `LCALL` instruction. Therefore, a `LCALL` instruction may be issued in any RAM page. After the `LCALL` instruction has completed, user code will be operating from the same RAM page as before the `LCALL` instruction was executed.

This instruction has a 16-bit unsigned address. A three-byte instruction format is used where the first byte is a full 8-bit opcode.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
LCALL	expr	$\text{ram}[\text{SP}] \leftarrow (\text{PC} + 3)[15:8]$ $\text{SP} \leftarrow \text{SP} + 1$ $\text{ram}[\text{SP}] \leftarrow (\text{PC} + 3)[7:0]$ $\text{SP} \leftarrow \text{SP} + 1$ $\text{PC} \leftarrow k, (0 \leq k \leq 65535)$	0x7C	13	3

Conditional Flags: CF Unaffected.
ZF Unaffected.

```
Example: 0000          _main:
          0000 7C 00 05  [13]  lcall SubFun
          0003 8F FC    [05]  jmp _main
```

Although in this example a full 16-bit address is not needed for the call to `SubFun`, the listing above shows that the `lcall` instruction is using a three byte format which accommodates the 16-bit absolute jump address of `0x0005`.

```
          0005
          0005          SubFun:
          0005 7F      [08]  ret
```


4.20 Long Jump

LJMP

Jump, unconditionally, to the unsigned address indicated by the instruction's argument. The `LJMP` instruction uses a three-byte instruction format to accommodate a full 16-bit argument. The first byte of the instruction is a full 8-bit opcode.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
LJMP	expr	$PC \leftarrow K, (0 \leq k \leq 65535)$	0x7D	7	3

Conditional Flags: CF Unaffected.
ZF Unaffected.

Example:

```

0000          _main:
0000 7D 00 03 [07]  ljmp SubFun

```

Although in this example a full 16-bit address is not needed for the jump to `SubFun` the listing above shows that the `ljmp` instruction is using a three byte format which accommodates the 16-bit absolute jump address of `0x0003`.

```

0003
0003          SubFun:
0003 7D 00 00 [07]  ljmp _main

```

Note that this instruction is jumping backwards, relative to the current `PC` value, and the address in the instruction is a positive number (`0x0000`). This is because the `ljmp` instruction uses an absolute address.

4.21 Move

MOV

Allows for a number of combinations of moves: immediate, direct, and indexed addressing are supported.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
MOV	X, SP	$X \leftarrow SP$	0x4F	4	1
MOV	A, expr	$A \leftarrow k$	0x50	4	2
MOV	A, [expr]	$A \leftarrow \text{ram}[k]$	0x51	5	2
MOV	A, [X+expr]	$A \leftarrow \text{ram}[X + k]$	0x52	6	2
MOV	[expr], A	$\text{ram}[k] \leftarrow A$	0x53	5	2
MOV	[X+expr], A	$\text{ram}[X + k] \leftarrow A$	0x54	6	2
MOV	[expr], expr	$\text{ram}[k_1] \leftarrow k_2$	0x55	8	3
MOV	[X+expr], expr	$\text{ram}[X + k_1] \leftarrow k_2$	0x56	9	3
MOV	X, expr	$X \leftarrow k$	0x57	4	2
MOV	X, [expr]	$X \leftarrow \text{ram}[k]$	0x58	6	2
MOV	X, [X+expr]	$X \leftarrow \text{ram}[X + k]$	0x59	7	2
MOV	[expr], X	$\text{ram}[k] \leftarrow X$	0x5A	5	2
MOV	A, X	$A \leftarrow X$	0x5B	4	1
MOV	X, A	$X \leftarrow A$	0x5C	4	1
MOV	A, reg[expr]	$A \leftarrow \text{reg}[k]$	0x5D	6	2
MOV	A, reg[X+expr]	$A \leftarrow \text{reg}[X + k]$	0x5E	7	2
MOV	[expr], [expr]	$\text{ram}[k_1] \leftarrow \text{ram}[k_2]$	0x5F	10	3
MOV	REG[expr], A	$\text{reg}[k] \leftarrow A$	0x60	5	2
MOV	REG[X+expr], A	$\text{reg}[X + k] \leftarrow A$	0x61	6	2
MOV	REG[expr], expr	$\text{reg}[k_1] \leftarrow k_2$	0x62	8	3
MOV	REG[X+expr], expr	$\text{reg}[X + k_1] \leftarrow k_2$	0x63	9	3

Conditional Flags: CF Unaffected.
ZF Set if A is the destination and the result is zero.

Example: `mov A, 0x01 ;accumulator will equal 1, ZF=0`
`mov A, 0x00 ;accumulator will equal 0, ZF=1`

4.22 Move Indirect, Post-Increment to Memory

MVI

A data pointer in RAM is used to move data between another RAM address and the Accumulator. The data pointer is incremented after the data transfer has completed.

For PSoC microcontrollers with more than 256 bytes of RAM, special page pointers are used to allow the MVI instructions to access data in remote RAM pages. Two page pointers are available, one for MVI read (MVI A, [[expr]++]) and another for MVI write (MVI [[expr]++], A). The data pointer is always found in the current RAM page. The page pointers determine which RAM page the data pointer's address will use. At the end of an MVI instruction, user code will be operating from the same RAM page as before the MVI instruction was executed.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
MVI	A, [[expr]++]	$A \leftarrow \text{ram}[\text{ram}[k]]$ $\text{ram}[k] \leftarrow \text{ram}[k] + 1$	0x3E	10	2
MVI	[[expr]++], A	$\text{ram}[\text{ram}[k]] \leftarrow A$ $\text{ram}[k] \leftarrow \text{ram}[k] + 1$	0x3F	10	2

Conditional Flags: CF Unaffected.
ZF Set if A is updated with zero.

Example 1:

```

mov  [10h], 4
mov  [11h], 3
mov  [EBh], 10h      ;initialize MVI read pointer to 10h
mvi  A, [EBh]        ;A=4, ram[EBh]=11h
mvi  A, [EBh]        ;A=3, ram[EBh]=12h

```

Example 2:

```

mov  [EBh], 10h      ;initialize MVI write pointer to 10h
mov  A, 8
mvi  [EBh], A        ;ram[10h]=8, ram[EBh]=11h
mov  A, 1
mvi  [EBh], A        ;ram[11h]=1, ram[EBh]=12h

```

Multi-Page Example 3:

```

mov  reg[CUR_PP], 2   ;set Current Page Pointer to 2
mov  [10h], 4         ;ram_2[10h]=4
mov  [11h], 3         ;ram_2[11h]=3
mov  reg[CUR_PP], 0   ;set Current Page Pointer back to 0
mov  reg[MVR_PP], 2   ;set MVI read RAM page pointer
mov  [EBh], 10h      ;initialize MVI read pointer to 10h
mvi  A, [EBh]        ;A=4, ram_0[EBh]=11h
mvi  A, [EBh]        ;A=3, ram_0[EBh]=12h

```

Multi-Page Example 4:

```

mov  reg[CUR_PP], 0   ;set Current Page Pointer to 0
mov  reg[MVW_PP], 3   ;set MVI write RAM page pointer
mov  [EBh], 10h      ;initialize MVI read pointer to 10h
mov  A, 8
mvi  [EBh], A        ;ram_3[10h]=8, ram_0[EBh]=11h
mov  A, 1
mvi  [EBh], A        ;ram_3[11h]=1, ram_0[EBh]=12h

```

4.23 No Operation

NOP

Performs no operation but consumes 4 CPU clock cycles. This is a one-byte instruction.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
NOP		None	0x40	4	1

Conditional CF Unaffected.
Flags: ZF Unaffected.

4.24 Bitwise OR

OR

Computes the logical OR for each bit position using both arguments. The result of the logical OR is placed in the corresponding bit position for the first argument.

The Carry Flag is only changed when the `OR F, expr` instruction is used. The Carry Flag will be set to the result of the logical OR of the Carry Flag at the beginning of instruction execution and the second argument's value at bit position 2 (i.e., `F[2]` and `expr[2]`).

For the `OR F, expr` instruction, the Zero Flag is handled the same as the Carry Flag in that it is changed as a result of the logical OR of the Zero Flag's value at the beginning of instruction execution, and the value of the second arguments value at bit position 1 (i.e., `F[1]` and `expr[1]`). However, for all other `OR` instructions the Zero Flag will be set or cleared based on the result of the logical OR operation. If the result of the `OR` instruction is that all bits are zero, the Zero Flag will be set; otherwise, the Zero Flag is cleared.

Note that `OR` (or `AND` or `XOR`, as appropriate) is a read-modify write instruction. When operating on a register, that register must be of the read/write type. Bitwise `OR` to a write only register will generate nonsense.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
OR	A, expr	$A \leftarrow A k$	0x29	4	2
OR	A, [expr]	$A \leftarrow A \text{ram}[k]$	0x2A	6	2
OR	A, [X+expr]	$A \leftarrow A \text{ram}[X + k]$	0x2B	7	2
OR	[expr], A	$\text{ram}[k] \leftarrow \text{ram}[k] A$	0x2C	7	2
OR	[X+expr], A	$\text{ram}[X + k] \leftarrow \text{ram}[X + k] A$	0x2D	8	2
OR	[expr], expr	$\text{ram}[k_1] \leftarrow \text{ram}[k_1] k_2$	0x2E	9	3
OR	[X+expr], expr	$\text{ram}[X + k_1] \leftarrow \text{ram}[X + k_1] k_2$	0x2F	10	3
OR	REG[expr], expr	$\text{reg}[k_1] \leftarrow \text{reg}[k_1] k_2$	0x43	9	3
OR	REG[X+expr], expr	$\text{reg}[X + k_1] \leftarrow \text{reg}[X + k_1] k_2$	0x44	10	3
OR	F, expr	$F \leftarrow F k$	0x71	4	2

Conditional Flags: CF Unaffected (unless `F` is destination).
ZF Set if the result is zero; cleared otherwise (unless `F` is destination).

Example 1: `mov A, 0x00`
`or A, 0xAA ;A=0xAA, CF=unchanged, ZF=0`

Example 2: `and F, 0x00`
`or F, 0x01 ;F=1 therefore CF=0, ZF=0`

4.25 Pop Stack into Register

POP

Removes the last byte placed on the stack and put it in the specified M8C register. The Stack Pointer is automatically decremented. The Zero Flag is set if the popped value is zero; otherwise, the Zero Flag is cleared. The Carry Flag is not affected by this instruction.

For PSoC devices with more than 256 bytes of RAM, the stack is confined to a single designated stack page defined by the value of the STK_PP Register. The M8C automatically selects the stack page as the source for the memory read during the POP instruction. Therefore, a POP instruction may be issued in any RAM page. After the POP instruction has completed, user code will be operating from the same RAM page as before the POP instruction was executed.

See the RAM Paging chapter of the *PSoC Technical Reference Manual (TRM)* for details.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
POP	A	$A \leftarrow \text{ram}[\text{SP} - 1]$ $\text{SP} \leftarrow \text{SP} - 1$	0x18	5	1
POP	X	$X \leftarrow \text{ram}[\text{SP} - 1]$ $\text{SP} \leftarrow \text{SP} - 1$	0x20	5	1

Conditional Flags: CF Unaffected.
ZF Set if A is updated to zero.

Example 1:

```

mov  A, 34
push A           ;top value of stack is now 34, SP+1
mov  A, 0        ;clear the Accumulator
pop  A           ;A=34, SP-1

```

Example 2:

```

mov  A, 34
push A           ;top value of stack is now 34, SP+1
pop  X           ;X=34, SP-1

```

4.26 Push Register onto Stack

PUSH

Transfers the value from the specified M8C register to the top of the stack, as indicated by the value of the CPU_SP register (SP) at the start of the instruction. After placing the value on the stack, the SP is incremented. The Zero Flag is set if the pushed value is zero, else the Zero Flag is cleared. The Carry Flag is not affected by this instruction.

For PSoC microcontrollers with more than 256 bytes of RAM, the stack is confined to a single designated stack page defined by the value of the STK_PP Register. The M8C automatically selects the stack page as the source for the memory write during the `PUSH` instruction. Therefore, a `PUSH` instruction may be issued in any PUSH page. After the `PUSH` instruction has completed, user code will be operating from the same RAM page as before the `PUSH` instruction was executed.

See the RAM Paging chapter of the *PSoC Technical Reference Manual (TRM)* for details.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
PUSH	A	ram[SP] ← A SP ← SP + 1	0x08	4	1
PUSH	X	ram[SP] ← X SP ← SP + 1	0x10	4	1

Conditional Flags: CF Unaffected.
ZF Unaffected.

Example 1: `mov A, 0x3E`
`push A` ;top value of stack is now 0x3E, SP+1

Example 2: `mov X, 0x3F`
`push X` ;top value of stack is now 0x3F, SP+1

4.27 Return

RET

The last two bytes placed on the stack are used to change the PC (CPU_PC register). The lower 8 bits of the PC are popped off the stack first, followed by the SP being decremented by one. Next, the upper 8 bits of the PC are popped off the stack, followed by a decrement of the SP. Neither Carry or Zero Flag is affected by this instruction.

For PSoC devices with more than 256 bytes of RAM, the stack is confined to a single designated stack page defined by the value of the STK_PP Register. The M8C automatically selects the stack page as the source for the pop during the RET instruction. Therefore, a RET instruction may be issued in any RAM page. After the RET instruction has completed, user code will be operating from the same RAM page as before the RET instruction was executed.

See the RAM Paging chapter of the *PSoC Technical Reference Manual (TRM)* for details.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
RET		$SP \leftarrow SP - 1$ $PC[7:0] \leftarrow \text{ram}[SP]$ $SP \leftarrow SP - 1$ $PC[15:8] \leftarrow \text{ram}[SP]$	0x7F	8	1

Conditional Flags: CF Unaffected.
ZF Unaffected.

```

Example:  0000          _main:
           0000  90 02    [11]  call   SubFun
           0002  40       [04]  nop
           0003  30       [04]  halt
           0004
           0004          SubFun:
           0004  40       [04]  nop
           0005  7F       [08]  ret

```

The RET instruction will set the PC to 0x0002, which is the starting address of the first instruction after the CALL.

4.28 Return from Interrupt

RETI

When the M8C takes an interrupt, three bytes are pushed onto the stack. One for CPU_F and two for the PC. When a RETI is executed, the last three bytes placed on the stack are used to change the CPU_F register and the CPU_PC register. The first byte removed from the stack is used to restore the CPU_F register. The SP (CPU_SP register) is decremented after the first byte is removed. The lower 8 bits of the PC are popped off the stack next, followed by the SP being decremented by one again. Finally, the upper 8 bits of the PC are popped off the stack, followed by a last decrement of the SP. The Carry and Zero Flags are updated with the values from the first byte popped off the stack.

For PSoC devices with more than 256 bytes of RAM, the stack is confined to a single designated stack page defined by the value of the STK_PP Register. The M8C automatically selects the stack page as the source for the pop during the RETI instruction. Therefore, an RETI instruction may be issued in any RAM page. After the RETI instruction has completed, user code will be operating from the same RAM page as before the RETI instruction was executed.

See the RAM Paging chapter of the *PSoC Technical Reference Manual (TRM)* for details.

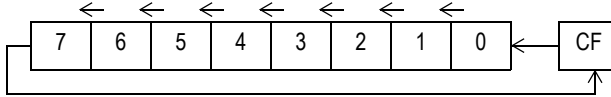
Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
RETI		$SP \leftarrow SP - 1$ $F \leftarrow \text{ram}[SP]$ $SP \leftarrow SP - 1$ $PC[7:0] \leftarrow \text{ram}[SP]$ $SP \leftarrow SP - 1$ $PC[15:8] \leftarrow \text{ram}[SP]$	0x7E	10	1

Conditional 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.

4.29 Rotate Left through Carry

RLC

Shifts all bits of the instruction's argument one bit 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.



Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
RLC	A	$ \begin{aligned} &CF \leftarrow A:7 \\ &A:7 \leftarrow A:6 \\ &A:6 \leftarrow A:5 \\ &A:5 \leftarrow A:4 \\ &A:4 \leftarrow A:3 \\ &A:3 \leftarrow A:2 \\ &A:2 \leftarrow A:1 \\ &A:1 \leftarrow A:0 \\ &A:0 \leftarrow CF \end{aligned} $	0x6A	4	1
RLC	[expr]	$ \begin{aligned} &CF \leftarrow ram[k]:7 \\ &ram[k]:7 \leftarrow ram[k]:6 \\ &ram[k]:6 \leftarrow ram[k]:5 \\ &ram[k]:5 \leftarrow ram[k]:4 \\ &ram[k]:4 \leftarrow ram[k]:3 \\ &ram[k]:3 \leftarrow ram[k]:2 \\ &ram[k]:2 \leftarrow ram[k]:1 \\ &ram[k]:1 \leftarrow ram[k]:0 \\ &ram[k]:0 \leftarrow CF \end{aligned} $	0x6B	7	2
RLC	[X+expr]	$ \begin{aligned} &CF \leftarrow ram[(X+k)]:7 \\ &ram[(X+k)]:7 \leftarrow ram[(X+k)]:6 \\ &ram[(X+k)]:6 \leftarrow ram[(X+k)]:5 \\ &ram[(X+k)]:5 \leftarrow ram[(X+k)]:4 \\ &ram[(X+k)]:4 \leftarrow ram[(X+k)]:3 \\ &ram[(X+k)]:3 \leftarrow ram[(X+k)]:2 \\ &ram[(X+k)]:2 \leftarrow ram[(X+k)]:1 \\ &ram[(X+k)]:1 \leftarrow ram[(X+k)]:0 \\ &ram[(X+k)]:0 \leftarrow CF \end{aligned} $	0x6C	8	2

Conditional Flags: CF Set if the MSB of the specified operand was set before the shift, cleared otherwise.

ZF Set if the result is zero; cleared otherwise.

Example:

```

and  F, 0xFB      ;clear carry flag
mov  A, 0x7F      ;initialize A with 127
rlc  A            ;A=0xFE, CF=0, ZF=0

```

4.30 Absolute Table Read

ROMX

Moves any byte from ROM (Flash) into the Accumulator. The address of the byte to be retrieved is determined by the 16-bit value formed by the concatenation of the CPU_A and CPU_X registers. The CPU_A register is the most significant byte and the CPU_X register is the least significant byte of the address. The Zero Flag is set if the retrieved byte is zero; otherwise, the Zero Flag is cleared. The Carry Flag is not affected by this instruction.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
ROMX		$t1 \leftarrow PC[7:0]$ $PC[7:0] \leftarrow X$ $t2 \leftarrow PC[15:8]$ $PC[15:8] \leftarrow A$ $A \leftarrow rom[PC]$ $PC[7:0] \leftarrow t1$ $PC[15:8] \leftarrow t2$	0x28	11	1

Conditional Flags: CF Unaffected.
 ZF Set if A is zero; cleared otherwise.

Example:

```

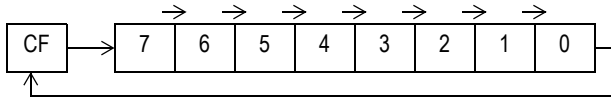
0000          _main:
0000 50 00    [04]  mov A, 00h
0002 57 08    [04]  mov X, 08h
0004 28      [11]  romx
0005 60 00    [05]  mov reg[00h], A
0007 40      [04]  nop
0008 30      [04]  halt
  
```

The ROMX instruction will read a byte from Flash at address 0x0008. The halt opcode is at address 0x0008; therefore, register 0x00 will receive the value 0x30.

4.31 Rotate Right through Carry

RRC

Shifts all bits of the instruction's argument one bit to the right. The Carry Flag is loaded into the most significant bit of the argument. Bit 0 of the argument is loaded into the Carry Flag.



Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
RRC	A	$A \leftarrow \begin{bmatrix} CF \leftarrow A:0, A:0 \leftarrow A:1, A:1 \leftarrow A:2 \\ A:2 \leftarrow A:3, A:3 \leftarrow A:4, A:4 \leftarrow A:5 \\ A:5 \leftarrow A:6, A:6 \leftarrow A:7, A:7 \leftarrow CF \end{bmatrix}$	0x6D	4	1
RRC	[expr]	$ram[k] \leftarrow \begin{bmatrix} CF \leftarrow ram[(k)]:0 \\ ram[k]:0 \leftarrow ram[k]:1 \\ ram[k]:1 \leftarrow ram[k]:2 \\ ram[k]:2 \leftarrow ram[k]:3 \\ ram[k]:3 \leftarrow ram[k]:4 \\ ram[k]:4 \leftarrow ram[k]:5 \\ ram[k]:5 \leftarrow ram[k]:6 \\ ram[k]:6 \leftarrow ram[k]:7 \\ ram[k]:7 \leftarrow CF \end{bmatrix}$	0x6E	7	2
RRC	[X+expr]	$ram[X+k] \leftarrow \begin{bmatrix} CF \leftarrow ram[(X+k)]:0 \\ ram[(X+k)]:0 \leftarrow ram[(X+k)]:1 \\ ram[(X+k)]:1 \leftarrow ram[(X+k)]:2 \\ ram[(X+k)]:2 \leftarrow ram[(X+k)]:3 \\ ram[(X+k)]:3 \leftarrow ram[(X+k)]:4 \\ ram[(X+k)]:4 \leftarrow ram[(X+k)]:5 \\ ram[(X+k)]:5 \leftarrow ram[(X+k)]:6 \\ ram[(X+k)]:6 \leftarrow ram[(X+k)]:7 \\ ram[(X+k)]:7 \leftarrow CF \end{bmatrix}$	0x6F	8	2

Conditional Flags: CF Set if LSB of the specified operand was set before the shift; cleared otherwise.

ZF Set if the result is zero; cleared otherwise.

Example 1:

```

or    F, 0x04    ;set carry flag
and   A, 0x00    ;clear the accumulator
rrc   A          ;A=0x80, CF=0, ZF=0

```

Example 2:

```

and   F, 0xFB    ;clear carry flag
mov   A, 0xFF    ;initialize A to 255
and   A, 0x00    ;make sure all flags are cleared
rrc   A          ;A=0x7F, CF=1, ZF=0

```

Example 3:

```

or    F, 0x04    ;set carry flag
mov   [0xEB], 0xAA ;initialize A to 170
rrc   [0xEB]     ;ram[0xEB]=0xD5, CF=1, ZF=0

```

4.32 Subtract with Borrow

SBB

Computes the difference of the two operands plus the carry value from the Flag register. The first operand's value is replaced by the computed difference. If the difference is less than '0' the Carry Flag is set in the Flag register. If the difference is zero, the Zero Flag is set in the Flag register; otherwise, the Zero Flag is cleared.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
SBB	A, expr	$A \leftarrow A - (K + CF)$	0x19	4	2
SBB	A, [expr]	$A \leftarrow A - (\text{ram}[k] + CF)$	0x1A	6	2
SBB	A, [X+expr]	$A \leftarrow A - (\text{ram}[X + k] + CF)$	0x1B	7	2
SBB	[expr], A	$\text{ram}[k] \leftarrow \text{ram}[k] - (A + CF)$	0x1C	7	2
SBB	[X+expr], A	$\text{ram}[X + k] \leftarrow \text{ram}[X + k] - (A + CF)$	0x1D	8	2
SBB	[expr], expr	$\text{ram}[k_1] \leftarrow \text{ram}[k_1] - (k_2 + CF)$	0x1E	9	3
SBB	[X+expr], expr	$\text{ram}[X + k_1] \leftarrow \text{ram}[X + k_1] - (k_2 + CF)$	0x1F	10	3

Conditional Flags: CF Set if treating the numbers as unsigned, the difference < 0; cleared otherwise.
ZF Set if the result is zero; cleared otherwise.

Example 1: `mov A, 0 ;set accumulator to zero`
`or F, 0x02 ;set carry flag`
`sbb A, 12 ;accumulator value is now 0xF3`

Example 2: `mov [0x39], 2 ;initialize ram[0x39]=0x02`
`mov [0x40], FFh ;initialize ram[0x40]=0xff`
`inc [0x40] ;ram[0x40]=0x00, CF=1`
`sbb [0x39], 0 ;ram[0x39]=0x01`

4.33 Subtract without Borrow

SUB

Computes the difference of the two operands. The first operand's value is replaced by the computed difference. If the difference is less than zero, the Carry Flag is set in the Flag register. If the difference is zero, the Zero Flag is set in the Flag register; otherwise, the Zero Flag is cleared.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
SUB	A, expr	$A \leftarrow A - K$	0x11	4	2
SUB	A, [expr]	$A \leftarrow A - \text{ram}[k]$	0x12	6	2
SUB	A, [X+expr]	$A \leftarrow A - \text{ram}[X + k]$	0x13	7	2
SUB	[expr], A	$\text{ram}[k] \leftarrow \text{ram}[k] - A$	0x14	7	2
SUB	[X+expr], A	$\text{ram}[X + k] \leftarrow \text{ram}[X + k] - A$	0x15	8	2
SUB	[expr], expr	$\text{ram}[k_1] \leftarrow \text{ram}[k_1] - k_2$	0x16	9	3
SUB	[X+expr], expr	$\text{ram}[X + k_1] \leftarrow \text{ram}[X + k_1] - k_2$	0x17	10	3

Conditional Flags:

CF Set if treating the numbers as unsigned, the difference < 0 ; cleared otherwise.

ZF Set if the result is zero; cleared otherwise.

Example 1:

```

mov  A, 0           ;set accumulator to zero
or   F, 0x04        ;set carry flag
sub  A, 12          ;accumulator value is now 0xF4

```

Example 2:

```

mov  [0x39], 2       ;initialize ram[0x39]=0x02
mov  [0x40], FFh     ;initialize ram[0x40]=0xff
inc  [0x40]          ;ram[0x40]=0x00, CF=1
sub  [0x39], 0       ;ram[0x39]=0x02

```

4.34 Swap

SWAP

Each argument is updated with the other argument's value. The Zero Flag is set if the Accumulator is updated with zero, else the Zero Flag is cleared. The `swap X, [expr]` instruction does not affect either the Carry or Zero Flags.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
SWAP	A, X	$t \leftarrow X$ $X \leftarrow A$ $A \leftarrow t$	0x4B	5	1
SWAP	A, [expr]	$t \leftarrow \text{ram}[k]$ $\text{ram}[k] \leftarrow A$ $A \leftarrow t$	0x4C	7	2
SWAP	X, [expr]	$t \leftarrow \text{ram}[k]$ $\text{ram}[k] \leftarrow X$ $X \leftarrow t$	0x4D	7	2
SWAP	A, SP	$t \leftarrow \text{SP}$ $\text{SP} \leftarrow A$ $A \leftarrow t$	0x4E	5	1

Conditional Flags: CF Unaffected.
ZF Set if Accumulator is cleared.

Example: `mov A, 0x30`
`swap A, SP ;SP=0x30, A equals previous SP value`

4.35 System Supervisor Call

SSC

Provides the method for users to access pre-existing routines in the Supervisory ROM. The supervisory routines perform various system-related functions. The CPU_PC and CPU_F registers are pushed on the stack prior to the execution of the supervisory routine. All bits of the Flag register are cleared before any supervisory routine code is executed; therefore, interrupts and page mode are disabled.

All supervisory routines return using the `RETI` instruction, causing the CPU_PC and CPU_F register to be restored to their pre-supervisory routine state.

Supervisory routines are device specific. Reference the data sheet for the device you are using for detailed information on the available supervisory routines.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
SSC		$\text{ram}[\text{SP}] \leftarrow \text{PC}[15:8]$ $\text{SP} \leftarrow \text{SP} + 1$ $\text{ram}[\text{SP}] \leftarrow \text{PC}[7:0]$ $\text{SP} \leftarrow \text{SP} + 1$ $\text{ram}[\text{SP}] \leftarrow \text{F}$ $\text{PC} \leftarrow 0x0000$ $\text{F} \leftarrow 0x00$	0x00	15	1

Conditional Flags:

CF	Unaffected.
ZF	Unaffected.

Example: The following example is one way to set up an SSC operation for the CY8C25xxx and CY8C26xxx PSoC devices. PSoC Designer uses the signature created by the following lines of code to recognize supervisory system calls and configures the In-Circuit Emulator for SSC debugging. It is recommended that users take advantage of the SSC Macro provided in PSoC Designer, to ensure that the debugger recognizes and therefore debugs supervisory operations correctly. See separate data sheets for complete device-specific options.

```

mov  X, SP           ;get stack pointers current value
mov  A, X            ;move SP to A
add  A, 3            ;add 3 to SP value
mov  [0xF9], A       ;store SP+3 value in ram[0xF9]=KEY2
mov  [0xF8], 0x3A    ;set ram[0xF9]=0x3A=KEY1
mov  A, 2            ;set supervisory function code = 2
SSC                  ;call supervisory function

```


4.36 Test for Mask

TST

Calculates a bitwise AND with the value of argument one and argument two. Argument one's value is not affected by the `TST` instruction. If the result of the AND is zero, the Zero Flag is set; otherwise, the Zero Flag is cleared. The Carry Flag is not affected by the `TST` instruction.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
TST	[expr], expr	ram[k ₁] & k ₂	0x47	8	3
TST	[X+expr], expr	ram[X + k ₁] & k ₂	0x48	9	3
TST	REG[expr], expr	reg[k ₁] & k ₂	0x49	9	3
TST	REG[X+expr], expr	reg[X + k ₁] & k ₂	0x4A	10	3

Conditional Flags: CF Unaffected.
ZF Set if the result of AND is zero; cleared otherwise.

Example:

```

mov  [0x00], 0x03
tst  [0x00], 0x02    ;CF=0, ZF=0 (i.e. bit 1 is 1)
tst  [0x00], 0x01    ;CF=0, ZF=0 (i.e. bit 0 is 1)
tst  [0x00], 0x03    ;CF=0, ZF=0 (i.e. bit 0 and 1 are 1)
tst  [0x00], 0x04    ;CF=0, ZF=1 (i.e. bit 2 is 0)

```

4.37 Bitwise XOR

XOR

Computes the logical XOR for each bit position using both arguments. The result of the logical XOR is placed in the corresponding bit position for the argument.

The Carry Flag is only changed when the `XOR F, expr` instruction is used. The CF will be set to the result of the logical XOR of the CF at the beginning of instruction execution and the second argument's value at bit position 2 (i.e., `F[2]` and `expr[2]`).

For the `XOR F, expr` instruction, the Zero Flag is handled the same as the Carry Flag in that it is changed as a result of the logical XOR of the Zero Flag's value at the beginning of instruction execution, and the value of the second argument's value at bit position 1 (i.e., `F[1]` and `expr[1]`). However, for all other XOR instructions, the Zero Flag will be set or cleared based on the result of the logical XOR operation. If the result of the XOR instruction is that all bits are zero, the Zero Flag will be set; otherwise, the Zero Flag is cleared. The Carry Flag is not affected.

Note that XOR (or AND or OR, as appropriate) is a read-modify write instruction. When operating on a register, that register must be of the read/write type. Bitwise XOR to a write only register will generate nonsense.

Instructions		Operation	Opcode	Cycles	Bytes
Mnemonic	Argument				
XOR	A, expr	$A \leftarrow A \oplus k$	0x31	4	2
XOR	A, [expr]	$A \leftarrow A \oplus \text{ram}[k]$	0x32	6	2
XOR	A, [X+expr]	$A \leftarrow A \oplus \text{ram}[X + k]$	0x33	7	2
XOR	[expr], A	$\text{ram}[k] \leftarrow \text{ram}[k] \oplus A$	0x34	7	2
XOR	[X+expr], A	$\text{ram}[X + k] \leftarrow \text{ram}[X + k] \oplus A$	0x35	8	2
XOR	[expr], expr	$\text{ram}[k_1] \leftarrow \text{ram}[k_1] \oplus k_2$	0x36	9	3
XOR	[X+expr], expr	$\text{ram}[X + k_1] \leftarrow \text{ram}[X + k_1] \oplus k_2$	0x37	10	3
XOR	REG[expr], expr	$\text{reg}[k_1] \leftarrow \text{reg}[k_1] \oplus k_2$	0x45	9	3
XOR	REG[X+expr], expr	$\text{reg}[X + k_1] \leftarrow \text{reg}[X + k_1] \oplus k_2$	0x46	10	3
XOR	F, expr	$F \leftarrow F \oplus k$	0x72	4	2

Conditional Flags: CF Unaffected (unless F is destination).
ZF Set if the result is zero; cleared otherwise (unless F is destination).

Example 1: `mov A, 0x00`
`xor A, 0xAA` ;A=0xAA, CF=unchanged, ZF=0

Example 2: `and F, 0x00` ;F=0
`xor F, 0x01` ;F=1 therefore CF=0, ZF=0

Example 3: `mov A, 0x5A`
`xor A, 0xAA` ;A=0xF0, CF=unchanged, ZF=0

5. Assembler Directives



This chapter covers all of the assembler directives currently supported by the ImageCraft Assembler. A description of each directive and its syntax will be given for each directive. ImageCraft Assembler directives are used to communicate with the ImageCraft Assembler and do not generate code. The directives allow a firmware developer to conditionally assemble source files, define symbolic equates for values, locate code or data at specific addresses, etc.

While the directives are often shown in all capital letters, the ImageCraft Assembler ignores case for directives and instructions mnemonics. However, the ImageCraft Assembler does consider case for user-defined symbols (i.e., labels). [Table 5-1](#) presents a summary of the assembler directives.

Table 5-1. ImageCraft Assembler Directives Summary

Symbol	Directive
AREA	Area
ASCIZ	NULL Terminated ASCII String
BLK	RAM Byte Block
BLKW	RAM Word Block
DB	Define Byte
DF	Define Floating-point Number
DS	Define ASCII String
DSU	Define UNICODE String
DW	Define Word
DWL	Define Word With Little Endian Ordering
ELSE	Alternative Result of IF Directive
ENDIF	End Conditional Assembly
ENDM	End Macro
EQU	Equate Label to Variable Value
EXPORT	Export
IF	Start Conditional Assembly
INCLUDE	Include Source File
.LITERAL, .ENDLITERAL	Prevent Code Compression of Data
MACRO	Start Macro Definition
ORG	Area Origin
.SECTION, .ENDSECTION	Section for Dead-Code Elimination
Suspend - OR F,0 Resume - ADD SP,0	Suspend and Resume Code Compressor

5.1 Area

AREA

Defines where code or data is located in Flash or RAM by the Linker. The Linker gathers all areas with the same name together from the source files, and either concatenates or overlays them, depending on the attributes specified. All areas with the same name must have the same attributes, even if they are used in different modules.

The following is a complete list of valid key words that can be used with the `AREA` directive:

RAM – Specifies that data is stored in RAM. Only used for variable storage. Commonly used with the `BLK` directive. Note that RAM AREAs are always overlay AREAs.

ROM – Specifies that code or data is stored in Flash.

ABS – Absolute, i.e., non-relocatable, location for code or data specified by the `ORG` directive. Default value of AREAs for type `ABS` or `REL` directives is not specified.

REL – Allows the Linker to relocate the code or data.

CON – Specifies that sequential AREAs follow each other in memory. Each AREA is allocated its own memory. The total size of the `AREA` directive is the sum of all AREA sizes. Default value of the AREAs for type `CON` or `OVR` directives is not specified.

OVR – Specifies that sequential AREAs start at the same address. This is a union of the AREAs. The total size of the `AREA` directive is the size of the largest area.

LIT - (New in ImageCraft Standard 7.0.5 and Professional 8.0.5) Specifies program literals that are stored in Flash. Typically contain lookup tables, strings, and so on. However this area can contain anything.

PROCLAB - (New in ImageCraft Standard 7.0.5 and Professional 8.0.5) Specifies sequences of entry points into the program code stored in Flash. Typically contains either function addresses or switch jump table entries. After optimizations, these entries are rewritten to point to the new addresses.

CODE - (New in ImageCraft Standard 7.0.5 and Professional 8.0.5) Specifies program instructions stored in Flash. The optimizer disassemble all areas that have instructions to perform optimizations. Literal data can be embedded within program code area by using the appropriate assembly directives. `LITERAL` / `.ENDLITERAL` (See “Prevent Code Compression of Data” section 5.15).

Note `lit`, `funcn_lit`, and `text` areas automatically have the `LIT`, `PROCLAB`, and `CODE` (new in ImageCraft Standard 7.0.5 and Professional 8.0.5) respectively applied to the area to maintain backward compatibility.

Directive	Arguments
AREA	<name> (< RAM ROM >, [ABS REL], [CON OVR], [LIT, PROCLAB, CODE])

Example: A code area is defined at address 2000.

```
AREA MyArea (ROM, ABS, CON)
    ORG 2000h
    _MyArea_start:
```

5.1.1 Code Compressor and the AREA Directive

With previous revisions of the ImageCraft compiler, calls to a text area from a non-text area were not allowed due to compatibility with the optimizer. Since the ImageCraft Standard 7.0.5 and ImageCraft Professional 8.0.5 compiler releases these calls are supported. Please verify which compiler you are using before implementing these directives.

5.2 NULL Terminated ASCII String

ASCIZ

Stores a string of characters as ASCII values and appends a terminating NULL (00h) character. The string must start and end with quotation marks ("").

The string is stored character by character in ASCII HEX format. The backslash character (\) is used in the string as an escape character. Non-printing characters, such as \n and \r, can be used. A quotation mark (") can be entered into a string using the backslash (\), a single quote (') as (\'), and a backslash (\) as (\\).

Directive	Arguments
ASCIZ	< "character string" >

Example: My"String\ is defined with a terminating NULL character.

```
MyString:  
    ASCIZ "My\"String\\"
```

5.3 RAM Block in Bytes

BLK

Reserves blocks of RAM in bytes. The argument is an expression, specifying the size of the block, in bytes, to reserve. The `AREA` directive must be used to ensure the block of bytes will reside in the correct memory location.

PSoC Designer requires that the `AREA bss` be used for RAM variables.

Directive	Arguments
BLK	< size >

Example: A 4-byte variable called MyVariable is allocated.

```
        AREA bss
MyVariable:
        BLK 4
```

5.4 RAM Block in Words

BLKW

Reserves a block of RAM. The amount of RAM reserved is determined by the size argument to the directive. The units for the size argument is words (16 bits).

PSoC Designer requires that the `AREA bss` be used for RAM variables.

Directive	Arguments
BLKW	< size >

Example: A 4-byte variable called MyVariable is allocated.

```
        AREA bss
MyVariable:
        BLKW 2
```


5.5 Define Byte

DB

Reserves bytes of ROM and assigns the specified values to the reserved bytes. This directive is useful for creating data tables in ROM.

Arguments may be constants or labels. The length of the source line limits the number of arguments in a DB directive.

Directive	Arguments
DB	< value1 > [, value2, ..., valuen]

Example: 3 bytes are defined starting at address 3000.

```
MyNum:  EQU 77h
         ORG 3000h
MyTable:
         DB 55h, 66h, MyNum
```

5.6 Define Floating-point Number

DF

Reserves four-byte pairs of ROM and assigns the specified values to each reserved pair. The format used is the IEEE-754 Single Format stored in big-endian format. This directive is useful for creating data tables in ROM.

Arguments must be constants. Only the length of the source line limits the number of arguments in a `DF` directive.

Directive	Arguments
DF	< value1 > [, value2, ..., valuen]

Example: MyTable:
 DF 1.2345, -1.07e-03f

5.7 Define ASCII String

DS

Stores a string of characters as ASCII values. The string must start and end with quotation marks (").

The string is stored character by character in ASCII HEX format. The backslash character (\) is used in the string as an escape character. Non-printing characters, such as \n and \r, can be used. A quotation mark (") can be entered into a string using the backslash (\), a single quote (') as (\'), and a backslash (\) as (\\).

The string is not null terminated. To create a null terminated string; follow the DS directive with a DB 00h or use ASCIIZ directive.

Directive	Arguments
DS	< "character string" >

Example: My"String\ is defined:
 MyString:
 DS "My\"String\\"

5.8 Define UNICODE String

DSU

Stores a string of characters as UNICODE values with little ENDIAN byte order. The string must start and end with quotation marks ("").

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. Non-printing characters, such as \n and \r, can be used. A quotation mark (") can be entered into a string using the backslash (\"), a single quote (') as (\'), and a backslash (\) as (\\).

Directive	Arguments
DSU	< "character string" >

Example: My"String\ is defined with little endian byte order.

```
MyString:
    DSU "My\"String\\"
```

5.9 Define Word, Big Endian Ordering

DW

Reserves two-byte pairs of ROM and assigns the specified words to each reserved byte. This directive is useful for creating tables in ROM.

The arguments may be constants or labels. Only the length of the source line limits the number of arguments in a `DW` directive.

Directive	Arguments
DW	< value1 > [, value2, ..., valuen]

Example: 6 bytes are defined starting at address 2000.

```
MyNum: EQU 3333h
        ORG 2000h
MyTable:
        DW 1111h, 2222h, MyNum
```

5.10 Define Word, Little Endian Ordering

DWL

Reserves two-byte pairs of ROM and assigns the specified words to each reserved byte, swapping the order of the upper and lower bytes.

The arguments may be constants or labels. The length of the source line limits the number of arguments in a `DWL` directive.

Directive	Arguments
DWL	< value1 > [, value2, ..., valuen]

Example: 6 bytes are defined starting at address 2000.

```
MyNum:  EQU 6655h
         ORG 2000h
MyTable:
         DWL 2211h, 4433h, MyNum
```

5.11 Equate Label

EQU

Assigns an integer value to a label. The label and operand are required for an `EQU` directive. The argument must be a constant or label or `"."` (the current PC). Each `EQU` directive may have only one argument and, if a label is defined more than once, an assembly error will occur.

To use the same equate in more than one assembly source file, place the equate in an `.inc` file and include that file in the referencing source files. Do not export equates from assembly source files, or the PSoC Designer Linker will resolve the directive in unpredictable ways.

Directive	Arguments
EQU	< label> EQU < value address >

Example: BITMASK is equated to 1Fh.

```
BITMASK: EQU 1Fh
```

5.12 Export

EXPORT

Designates that a label is global and can be referenced in another file. Otherwise, the label is not visible to another file. Another way to export a label is to end the label definition with two colons (::) instead of one.

Directive	Arguments
EXPORT	EXPORT < label >

Example:

```
Export MyVariable
    AREA bss
MyVariable:
    BLK 1
```


5.13 Conditional Source

IF, ELSE, ENDIF

All source lines between the `IF` and `ENDIF` (or `IF` and `ELSE`) directives are assembled if the condition is true. These statements can be nested.

`ELSE` delineates a “not true” action for a previous `IF` directive.

`ENDIF` finishes a section of conditional assembly that began with an `IF` directive.

Directive	Arguments
IF ELSE ENDIF	value

Example: Sections of the source code are conditional.

```

Cond1: EQU 1
Cond2: EQU 0
      ORG 1000h
      IF (Cond1)
      ADD A, 33h
      IF (Cond2)
      ADD A, FFh
      ENDIF ;Cond1
      NOP ;Cond1
      ELSE
      MOV A, FFh
      ENDIF ;Cond2
// The example creates the following code
      ADD A, 33h
      NOP

```

5.14 Include Source File

INCLUDE

Used to add additional source files to the file being assembled. When an `INCLUDE` directive is encountered, the ImageCraft Assembler reads in the specified source file until either another `INCLUDE` directive is encountered or the end of file is reached. If additional `INCLUDE` directives are encountered, additional source files are read in. When an end of file is encountered, the Assembler resumes reading the previous file.

Specify the full (or relative) path to the file if the source file does not reside in the current directory.

Directive	Arguments
INCLUDE	< file name >

Example: Three files are included into the source code.

```
INCLUDE "MyInclude1.inc"  
INCLUDE "MyIncludeFiles\MyInclude2.inc"  
INCLUDE "C:\MyGlobalIncludeFiles\MyInclude3.inc"
```

5.15 Prevent Code Compression of Data .LITERAL, .ENDLITERAL

Used to avoid code compression of the data defined between the `.LITERAL` and `.ENDLITERAL` directives. For the code compressor to function, all data defined in ROM with the `ASCIZ`, `DB`, `DS`, `DSU`, `DW`, or `DWL` directives must use this directive. The `.LITERAL` directive must be followed by an exported global label. The `.ENDLITERAL` directive resumes code compression.

Directive	Arguments
<code>.LITERAL</code> <code>.ENDLITERAL</code>	< none >

Example: Code compression is suspended for the data table.

```
Export DataTable
.LITERAL
DataTable:
DB 01h, 02h, 03h
.ENDLITERAL
```

5.16 Macro Definition

MACRO, ENDM

Used to specify the start and end of a macro definition. The lines of code defined between a `MACRO` directive and an `ENDM` directive are not directly assembled into the program. Instead, it forms a macro that can later be substituted into the code by a macro call. The following `MACRO` directive is 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 and labels that are referenced from outside the macro.

Labels in macros have to be local to each invocation of the macro, for example:

```
macro xyz
mov a,2
loop: mov a,x
jnz loop
endmacro
```

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.

Directive	Arguments
MACRO ENDM	< name >< arguments >

Example: A `MACRO` is defined and used in the source code.

```
MACRO MyMacro
ADD A, 42h
MOV X, 33h
ENDM
// The Macro instructions are expanded at address 2400
ORG 2400h
MyMacro
```

5.17 Area Origin

ORG

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 areas with the ABS mode.

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

Directive	Arguments
ORG	< address >

Example: The bytes defined after the `ORG` directive are at address 1000.

```
ORG 1000h
DB 55h, 66h, 77h
```

5.18 Section for Dead-Code Elimination .SECTION, .ENDSECTION

Allows the removal of code specified between the `.SECTION` and `.ENDSECTION` directives. The `.SECTION` directive must be followed by an exported global label. If there is no call to the global label, the code will be eliminated and call offsets will be adjusted appropriately. The `.ENDSECTION` directive ends the dead-code section. Note that use of this directive is not limited to removing dead code.

PSoC Designer takes care of dead code. Check the “Enable Elimination of un-used User Modules (area) APIs” field under the Project > Settings > Compiler tab. If you check this field upon a build, the system will go in and remove all dead code from the APIs in an effort to free up space.

Directive	Arguments
<code>.SECTION</code> <code>.ENDSECTION</code>	< none >

Example: The section of code is designated as possible dead code.

```
Export Counter8_1_WriteCompareValue
.SECTION
Counter8_1_WriteCompareValue:
    MOV     reg[Counter8_1_COMPARE_REG], A
    RET
.ENDSECTION
```

5.19 Suspend/Resume Code Compressor OR F,0; ADD SP,0

Used to prevent code compression of the code between the `OR F,0` and `ADD SP,0` instructions. The code compressor may need to be suspended for timing loops and jump tables. If the `JACC` instruction is used to access fixed offset boundaries in a jump table, any `LJMP` and/or `LCALL` instruction entries in the table may be optimized to relative jumps or calls, changing the proper offset value for the `JACC`. A `RET` or `RETI` instruction will resume code compression if it is encountered before an `ADD SP,0` instruction. These instructions are defined as the macros `Suspend_CodeCompressor` and `Resume_CodeCompressor` in the file `m8c.inc`.

Directive	Arguments
<code>OR F,0</code> <code>ADD SP,0</code>	< none >

Example: Code compression is suspended for the jump table.

```
OR F,0
MOV A, [State]
JACC StateTable
StateTable:
LJMP State1
LJMP State2
LJMP State3
ADD SP,0
```


6. Large Memory Model



This chapter presents a detailed overview of the memory architecture designed to accommodate PSoC® 1 devices with a RAM space larger than 256 bytes. A thorough discussion of programming requirements and challenges are presented along with code examples for reading from and writing to the expanded RAM area.

Several PSoC 1 devices have more than 256 bytes of RAM. Because the M8C Sore features an 8-bit address bus, direct addressing of all the expanded memory addresses is impossible and a paging scheme is needed to handle more than 256 bytes of RAM. This section discusses a programming model that takes into account the paging architecture of these 'Large Memory Model' (LMM) PSoC devices.

This chapter is intended for use by anyone planning to write firmware for large memory devices (with more than 256 bytes of RAM). It is most useful to those who are developing in the assembly language, where you have to deal with the paging architecture. The C compiler handles most aspects of the Large Memory Model.

6.1 Large Memory Model Registers

M8C instructions use 8-bit operands for RAM addresses and therefore are limited to accessing a 256-byte range. In parts with greater than 256 bytes of RAM, the memory is broken up into several pages, each containing 256 bytes. The page to access when performing an operation is then left to a number of registers that are added to Large Memory Model devices. There are multiple registers, each of which applies to a specific set of instructions. The registers are listed and described in the following sections.

6.1.1 Current Page Pointer (CUR_PP)

The Current Page Pointer specifies the page on which direct memory instructions operate. Direct memory instructions are those where a variable address is given explicitly as an operand to the instruction. Refer to Page Mode Bits in section 6.1.5 for exceptions to CUR_PP usage.

Examples of instructions using the Current Page Pointer:

```
mov  [var_name], A
add  A, [var_name]
```

Usage:

```
mov  A, 0x55
mov  reg[CUR_PP], 0x01
//Place 0x55 at address 0x0100
mov  [0x00], A
mov  reg[CUR_PP], 0x02
//Place 0x55 at address 0x0200
mov  [0x00], A
```

6.1.2 Stack Page Pointer (STK_PP)

The Stack Page Pointer specifies the page on which stack instructions operate. Stack operations are those during which data is pushed and popped from the stack. Refer to Page Mode Bits in section 6.1.5 for an exception to STK_PP usage.

Examples of instructions using the Stack Page Pointer:

```
push A
pop X
call Foo
reti
```

Usage:

```
mov A, 0x55
//Set stack page to page 7
mov reg[STK_PP], 0x07
//Place 0x55 on page 7 at SP
push A
//Call function
call Foo
```

6.1.3 Index Page Pointer (IDX_PP)

The Index Page Pointer specifies the page on which Indexed Mode Memory instructions operate. Indexed Mode Memory instructions are those for which the Index register is used as a pointer to the data. Refer to Page Mode Bits in section 6.1.5 for an exception to IDX_PP usage.

Examples of instructions using the Index Page Pointer:

```
mov [X+4], A
add A, [X]
```

Usage:

```
mov A, 0x55
//Set Index page to MSB of var address
mov reg[IDX_PP], >baArray
//Set X register to LSB of var address
mov X, <baArray
//Place 0x55 in variable
mov [X], A
//get next data from array, place in A
mov A, [X+1]
```

6.1.4 MVI Page Pointers (MVR_PP, MVW_PP)

The Move Indexed Read and Write Page Pointers specify the page(s) on which the MVI instructions operate. The page pointers for the source for read operations and the destination for write operations are contained in the MVR_PP or MVW_PP registers, respectively. The page pointer to the index variable is located in the CUR_PP register. The following example shows the relationship between operations and the page pointers. For more information on the MVI instruction, refer to section 4.22.

```
//set current page to page 1
mov reg[CUR_PP], 0x01
//set the mvi read page to page 2
mov reg[MVR_PP], 0x02
//set the mvi write page to page 3
```

```

mov reg[MVW_PP], 0x03
//set the pointer variable on page 1 (assigned in the CUR_PP) to 0x40
mov [pointer], 0x40
//read from RAM address 0x0240
mvi A, [pointer]
//write to RAM address 0x0340
mvi [pointer], A

```

Page Mode Bits

The Flag register contains two bits that determine the relationship between certain instructions and the page on which they operate. Instructions effected by the Page Mode bits are the direct memory instructions and the indexed memory instructions. Table 1 lists the different modes offered and their effect.

Since the page mode bits can be modified at any time, sometimes it is beneficial to change page modes to perform certain tasks. One example is accessing the stack through indexed instructions by using Mode 3 addressing (mode bits set to 0x11). This can be useful when accessing a particular memory location in the stack without being confined to PUSH and POP operations. Several other common page mode examples are presented later in this chapter.

Table 6-1. Page Mode Bits

Mode	Mode #	Direct Memory Instructions	Indexed Memory Instructions
0	0	Operate on Page 0 Only	Operate on Page 0 Only
0	1	Operate on Page 0 Only	Operate on Page Specified by STK_PP
1	0	Operate on Page Specified by CUR_PP	Operate on Page Specified by IDX_PP
1	1	Operate on Page Specified by CUR_PP	Operate on Page Specified by STK_PP

6.2 Large Memory Model Firmware

To handle the Large Memory Model, a number of new concepts and conventions are added to the PSoC programming paradigm. Managing different modes and pages within the complex code and, more importantly, across the function calls, can lead to unwanted behavior without a properly defined set of conventions. Some of these conventions take the form of macros for common operations. Others are constants used by the code to determine proper behavior. Some are just policies that are to be observed by the User Module APIs, interrupts, and compiler.

6.2.1 Memory.inc

The memory.inc file, which is generated during compilation, contains all the constants and macros required to support the Large Memory Model. Directly modifying memory.inc has no affect because it is recreated and overwritten during every build. Details on the macros contained in memory.inc are provided in section 6.7.

6.2.2 Native Paging Mode

The Native Paging Mode concept is rooted in the realization that every application has a preferred method of executing Indexed Mode Memory accesses. There are two Native Paging Modes from Table 1 to choose from: Mode 2, which force Indexed Mode Memory operations to occur on the Indexed Page (specified by IDX_PP), and Mode 3, which force Indexed Mode Memory operations to occur on the stack page (specified by STK_PP). Mode 0 and Mode 1 are not used as Native Paging Modes. The boot.asm code guarantees that the Native Paging Mode is selected before entering the main user code. From that point forward, all the application programming interface (API) functions

expect to be in the Native Paging Mode when called. If the API changes the Page Mode at any time during execution, the Native Paging Mode should be restored before returning.

The ImageCraft C compiler operates primarily in Page Mode 3 and is the default Native Paging Mode for projects. After the code is compiled, all references to the Native Paging Mode refer to the selection made in *local.mk*.

The default paging mode can be changed to Mode 2 by modifying the *local.mk* file. The *local.mk* either overrides existing values in the *project.mk* or adds additional options to be processed by the build system. The default value for Native Paging Mode is identified in *project.mk* by the following directive:

```
SYSTEM_TOOLS=1
```

To change the Native Paging Mode to *SYSTEM_IDXPG_TRACKS_IDX_PP* (Mode 2), assign the *SYSTEM_TOOLS* value as 0. This is done in *local.mk*. To open *local.mk* in PSoC Designer, go to Project >> Create *local.mk* file. This opens *local.mk* and creates it if one does not exist. Now, add the following line to the file to set Native Paging Mode to *SYSTEM_IDXPG_TRACKS_IDX_PP*:

```
SYSTEM_TOOLS=0
```

The selection made in *local.mk* propagates into the code through two assembly language equates listed in *memory.inc*:

```
SYSTEM_IDXPG_TRACKS_STK_PP: equ 1
```

```
SYSTEM_IDXPG_TRACKS_IDX_PP: equ 0
```

In any given project, only one of these constants is set to '1' while the other is set to '0'.

If *SYSTEM_IDXPG_TRACKS_STK_PP* is set to '1', Page Mode 3 is selected.

If *SYSTEM_IDXPG_TRACKS_IDX_PP* is set to '1', Page Mode 2 is selected

6.2.3 LMM Register Policies

A simple set of policies defines what the expectations are when calling and returning from functions. We have already seen one of these policies: Page Mode bits are expected to be in Native Paging Mode upon entry and exit from any API function. The remaining policies deal with the various Page Pointer Registers.

Table 6-2. LMM Registers

Register	Description
CUR_PP	Except for in an ISR, these registers are volatile and can be changed by functions without preserving and restoring their original values.
IDX_PP	
MVR_PP	
MVW_PP	
STK_PP	This register is managed and its value should not be changed by functions or ISRs.

6.2.4 API Classes and Page Pointer Usage

User Module APIs vary in functionality from simple to very complex; their effects on LMM registers vary significantly. Because most of the LMM registers are volatile, the need to restore their value every time a function is called introduces much overhead. This is one of the primary advantages of classifying the APIs. The class of the API communicates valuable information about whether the volatile LMM registers can be modified in the function or not. In some cases, the API class alone is not enough information to understand all of the Page Pointer usage in a function. Therefore, each API

function has its Page Pointer impact documented in code comments and in the User Module data sheet.

There are four classes of API functions, each differentiated by the type of instructions they employ.

Class 1

Class 1 functions are restricted to using stack functions such as PUSH, POP, CALL and RET. Class 1 functions do not modify any of the LMM registers. Therefore, there is no need to preserve them when calling such functions.

Class 2

Class 2 functions employ indexed instructions that use the stack page, when the device is in Page Mode 1 or 3. Class 2 functions do not modify any of the LMM registers. There is no need to preserve page mode registers when calling Class 2 functions.

Class 3

Class 3 functions employ indexed instructions that use data on any page, when the device is in Page Mode 2. A Class 3 function usually modifies the IDX_PP register and it is recommended that IDX_PP is preserved before calling the function and restored when returning.

Class 4

Class 4 functions are not restricted to any one set of instructions but may use all of them. These functions almost always change the CUR_PP register and in some instances, change the IDX_PP register. Rarer still are instances where the MVR_PP and the MVW_PP are modified.

By referring to the API code comments or the User Module data sheet, more efficient code can be developed that only saves and restores the page pointer registers that are used.

An important point is that API function implementations do change from version to version. This may result in an API function using more page pointer registers in later versions. Assembly language programmers must verify that no such change occurs when upgrading.

6.2.5 Fastcall16

Fastcall16 is an argument-passing convention used by the C compiler to determine the method by which arguments are passed to and returned from the API functions. Table 3 describes the fastcall16 convention for passing arguments to a function; Table 4 describes the fastcall16 convention for returning values from a function. For more information on fastcall16 and interfacing C with assembly, refer to the C Compiler User's Guide.

Table 6-3. Fastcall16 Conventions for Argument Passing

Argument Type	Register	Argument Register
Single Byte	A	The argument is passed in A
Two Single Byte	A, X	The first argument is passed in A, the second in X.
Double Byte	X, A	The MSB is passed in X, the LSB in A
Pointer	A, X	The MSB is passed in A, the LSB in X
All Others	None	Arguments are stored on the stack in standard byte order and in reverse order of appearance. In other words, the MSB of the last parameter is pushed first and the LSB of the first parameter is pushed last.

Table 6-4. Fastcall16 Conventions for Return Values

Return Type	Return Register	Comment
Single Byte	A	The argument is returned in A.
Double Byte	X, A	The MSB is passed in X, the LSB in A.
Pointer	A, X	The MSB is passed in A, the LSB in X.
All Others	None	Use a pass-by-reference parameter or global variable instead of returning arguments longer than 16 bits.

6.3 Going from Small to Large Memory Model

For the assembly language programmer, the Large Memory Model presents a number of additional requirements that did not exist before. For the most part, they are simple to deal with and further simplified by the macros made available in `memory.inc`. This section discusses these challenges and demonstrates how to handle them so that programming in the Large Memory Model is successful.

One of the basic requirements for working within the Large Memory Model is the management of pages and modes. When performing any instruction that writes to or reads from a memory location, it is important to know on which page this variable lies and to make sure that the relevant page pointer is correctly set to that page. This section demonstrates the requirements for Large Memory Model design through the use of simple examples. In each example, a Small Memory Model implementation is shown followed by a Large Memory Model implementation. For the first group of examples, it is assumed that the reader is not inside an ISR and that the Page Mode bits are in Native Paging Mode.

6.3.1 Example 1 - Simple Data Movement

This example demonstrates moving data from one variable to another.

Small Memory Model:

```
//Move data from variable to A
mov A, [bData1]

//Move data from A to variable
mov [bData2], A
```

Large Memory Model:

```
//Put MSB of var address in CUR_PP
RAM_SETPAGE_CUR >bData1
//Move data from variable to A
```

```

mov A, [bData1]
//Put MSB of var address in CUR_PP
RAM_SETPAGE_CUR >bData2
//Move data from A to variable
mov [bData2], A

```

Note the primary difference is that the Current Page Pointer must be set before every move. The previous code uses the macro RAM_SETPAGE_CUR (from memory.inc) to set the Current Page Pointer (CUR_PP) to the MSB of the address variable. This ensures that the correct page is used no matter where the variable is located.

The CUR_PP is set every time a memory operation is performed because there is no guarantee that the variables are on the same page. Some techniques, which are discussed later, can be used to guarantee that a group of variables is allocated to the same page. If both variables are on the same page, there is a simpler implementation for the same data movement.

Large Memory Model with Variables on Same Page:

```

//Put MSB of var address in CUR_PP
RAM_SET_PAGE_CUR >bData1
//Move data from variable to A
mov A, [bData1]
//Move data from A to variable
mov [bData2], A

```

6.3.2 Example 2 - Moving Stack Data

In this example, some variables are pushed to the stack and then one of them is retrieved through an Indexed Memory instruction.

Small Memory Model:

```

//A gets 0x55
mov A, 0x55
//0x55 gets pushed on stack
push A
//A gets 0x66
mov A, 0x66
//0x66 gets pushed on stack
push A
//X gets Stack Pointer
mov X, SP
//A gets 0x55
mov A, [X-2]
//bData1 gets 0x55
mov [bData1], A

```

In the earlier example, the small memory model is used so no page management is required (all operations occur on the same page). The SP always holds the location into which a new PUSH can be done, so in order to access 0x55, the SP value should be decremented by 2 to read the required value. So, [X - 2] is used when you want to copy 0x55 to the Accumulator.

With large memory model programming, the Page Mode bits become an important concern.

First, let's examine the same example with large memory model programming operating with the Native Paging Mode set to Paging Mode 3 (the default paging mode for the ImageCraft C compiler).

Large Memory Model in Native Paging Mode 3:

```

//A gets 0x55
mov  A, 0x55
//0x55 gets pushed on stack
push A
//A gets 0x66
mov  A, 0x66
//0x66 gets pushed on stack
push A
//X gets Stack Pointer
mov  X, SP
//A gets 0x55
mov  A, [X-2]
//Put MSB of variable bData1 address in CUR_PP. This is required because the //
variable page is different from the stack page
RAM_SETPAGE_CUR >bData1
//bData1 gets 0x55

mov  [bData1], A

```

Note the primary difference from the Small Memory Model example is that the Current Page Pointer (CUR_PP) must be set before moving the data from the accumulator into the variable. The index page (IDX_PP) does not need to be set because in Paging Mode 3, Indexed operations automatically occur on the current stack page.

Next, let's examine the same example again with the Native Paging Mode using Paging Mode 2, where Indexed instructions operate on the page specified by IDX_PP.

Large Memory Model in Native Paging Mode 2:

```

//A gets 0x55
mov  A, 0x55
//0x55 gets pushed on stack
push A
//A gets 0x66
mov  A, 0x66
//0x66 gets pushed on stack
push A
//X gets Stack Pointer
mov  X, SP
//Temporarily set page mode to 3 so that Indexed addressing points to stack
RAM_X_POINTS_TO_STACKPAGE
//A gets 0x55
mov  A, [X-2]
//Return to Native Paging Mode.
RAM_RESTORE_NATIVE_PAGING
//put MSB of var address in
RAM_SETPAGE_CUR >bData1
//bData1 gets 0x55

CUR_PP mov  [bData1], A

```

In this case, before the indexed operation can be used to access the stack, the paging mode must be temporarily changed to allow the indexed instruction operate on the stack page. To do this, the macro `RAM_X_POINTS_TO_STACKPAGE` (from `memory.inc`) is used. This macro simply sets the

paging mode to Paging Mode 3, which allows indexed instructions to operate on the stack page (STK_PP). After the indexed instruction is complete, the Native Paging Mode is restored by using the RAM_RESTORE_NATIVE_PAGING macro (also in memory.inc).

6.3.3 Example 3 - Array Access

This example demonstrates how to access an array. Array access is usually done through Indexed Mode Memory instructions, so in Large Memory Model programming the Index Page Pointer (IDX_PP) must be managed.

The example loops several times; calling a function and placing the returned value into an array. The array is filled from the end to the beginning to improve efficiency (a decrement and jump zero is more efficient than an increment, compare, and jump).

Small Memory Model:

```
//Setup count to the size of array
mov X, ARRAY_SIZE+1
begin:
//Decrement count
dec X
//Check if count ended
jz .end
//Call Class 3 function
call GetData
//Store data in array
mov [X + baVar], A
//Loop to beginning
jmp .begin
.end:
```

In the Small Memory Model, no page management is required.

With Large Memory Model programming, the Index Page Pointer must be set before accessing the array, as shown in the following code where the Native Paging Mode is set to Paging Mode 2 (Indexed instructions operate on the page indicated by IDX_PP).

Large Memory Model in Native Paging Mode 2:

```
//Setup count to the size of array
mov X, ARRAY_SIZE+1
begin:
//Decrement count
dec X
//Check if count ended
jz .end
//Call Class 3 function
call GetData
//Set the Indexed ops to right page
RAM_SETPAGE_IDX >baVar
//Store data in array
mov [X + baVar], A
//loop to beginning
jmp .begin
.end:
```

Note Since GetData is a Class 3 function (that is, it modifies the IDX_PP register), the IDX_PP register cannot simply be set once at the beginning, but must be set every time through the loop after the function is called.

A more difficult case is where the Native Paging Mode is set to Paging Mode 3, where Indexed instructions operate on the stack page pointer (STK_PP). This requires a temporary change from the Native Paging Mode so that the designer can access the page specified by IDX_PP. This is demonstrated in the following code.

Large Memory Model in Native Paging Mode 3:

```
//Setup count to size of array
mov X, ARRAY_SIZE+1
.begin:
//Decrement count
dec X
//Check if count ended
jz .end
//Call Class 3 function
call GetData
//Change to paging mode 2
RAM_X_POINTS_TO_INDEXPAGE
//Set Index pointer to right page
RAM_SETPAGE_IDX >baVar
//Store data in array
mov [X + baVar], A
//Return to native paging
RAM_RESTORE_NATIVE_PAGING
//loop to beginning
jmp .begin
.end:
```

In this case, the paging mode must be temporarily set to Paging Mode 2 before the returned variable can be moved to the array. This is done using the RAM_X_POINTS_TO_INDEXPAGE macro from memory.inc, which changes the paging mode so that indexed instructions operate on the page indicated by IDX_PP rather than the stack page. After the paging mode has been changed, the proper page is loaded into the IDX_PP register and the data is moved into the array. Lastly, the Native Paging Mode is restored before continuing.

Note It is imperative to restore the Native Paging Mode before calling other functions (especially User Module API functions which assume the Native Paging Mode as active).

6.3.4 Example 4 - MVI Instructions

This example demonstrates how to manage the MVR_PP and MVW_PP page pointers when performing an indirect move operation (MVI) with the large memory model. MVI instructions are used when reading/writing an absolute value to/from a particular address. In the small memory model, where all the variables reside in one page, the use of MVI instructions is direct.

The following is a small example of how to "read" a value from address 0x40 to the accumulator and to "write" the value to the address 0x50 using MVI instruction using the small memory model.

Small Memory Model:

```
//Set the pointer variable to 0x40. The address to read the value
mov [pointer], 0x40
//Read from RAM address 0x40 using the variable
mvi A, [pointer]
```

```
//Set the pointer variable to 0x50. The address to write the value
mov [pointer], 0x50
//Write to RAM address 0x50
mvi [pointer], A
```

When LMM is used, there is no guarantee that all the variables reside in the same page. With LMM, you have to set the page numbers for the data read and write. As explained earlier, the LMM registers include the MVR_PP, which is the Read Page register for the MVI instructions, to read from a particular address in the page number stored in MVR_PP. The MVW_PP is the Write Page register for the MVI instructions to write to a particular address in the page number stored in MVW_PP.

Now, modify the same example given previously to "read" a value from address 0x240 (address 0x40 in page 2) and "write" a value to address 0x350 (address 0x50 in page 3). Assume that the intermediate variable pointer, which is used to store the address value, is in Page 1.

Large Memory Model:

```
//Set current page to pointer MSB
RAM_SETPAGE_CUR >pointer
//Set the pointer variable
//(assigned in the CUR_PP) to 0x40
mov [pointer], 0x40
//Set the mvi read page to page 2
mov reg[MVR_PP], 0x02
//Read from RAM address 0x0240. The MVI takes the lower 8 byte of the address
//from the "pointer" variable and the upper byte of the address (page info)
//from the MVR_PP register
mvi A, [pointer]

//Set the mvi write page to page 3
mov reg[MVW_PP], 0x03
//Set the pointer variable on page 1 (assigned in the CUR_PP) to 0x50 to
//perform the write operation
mov [pointer], 0x50

//Write to RAM address 0x0350. The MVI takes the lower byte of address from
//the "pointer" variable. It takes the upper byte of address (page info) the
//MVW_PP register
mvi [pointer], A
```

In this example, several page pointers must be set. First, the current page pointer (CUR_PP) is setup to ensure pointer is set to 0x40. Next, the MVI read page pointer (MVR_PP) is set to point to page 2, since our target address to read is 0x240. After the MVI read is done, the MVI write page pointer (MVW_PP) is set to point to page 3, since our target address to write is at address 0x350.

6.4 Calling Functions

The Large Memory Model introduces one concern with regard to calling functions. Since the policy for the Page Pointer registers states that they are volatile and subject to change, calling routines may alter the values that are in CUR_PP, IDX_PP, MVR_PP, and MVW_PP. This section discusses how to handle functions that modify Page Pointer registers. Argument passing with fastcall16 and how that impacts legacy code is also discussed.

Functions that modify Page Pointer registers fall into Classes 3 (which modify only IDX_PP) and 4 (which modify any Page Pointer registers). While understanding the class concept is useful in developing assembly code that is exposed to C language, it is less important in cases when you wish only to call existing functions. The primary concern is which Page Pointer registers are modified by the function. This information is clearly documented in the data sheet and in the .asm file that includes the API functions being called.

In the datasheet, the API section has a listing of all the User Module API functions. For each function there is a "Side Effects" section that identifies, if applicable, which Page Pointer registers are modified. This information is mirrored in the comment block immediately preceding each API function in the User Module's .asm file.

The following code demonstrates how to call a Class 3 function, which modifies IDX_PP. The code first preserves the IDX_PP register, using the REG_PRESERVE macro before loading a pointer into A and X, which is used by the function. After calling the function, the REG_RESTORE macro is used to restore the IDX_PP register to its original value., loads a pointer into A and X and then calls a function that modifies the IDX_PP register.

Calling a Class 3 Function:

```
//Save the IDX_PP register
REG_PRESERVE IDX_PP
//Pass a pointer to the function
mov A, 0x04
mov X, 0x33
//Call the function
call SomeClass3Func
//Pop the old value and put in IDX_PP
REG_RESTORE IDX_PP
```

In the earlier example, IDX_PP is preserved at the start to make sure the value of A is not corrupted. Preserving and restoring the register may not be necessary. In most cases, you can simply set the IDX_PP to the correct value after returning from the call if you are aware of the correct page. This saves the overhead of saving and restoring the IDX_PP value when calling a Class 3 function.

Calling a Class 3 Function when the IDX_PP Value is known:

```
//Pass a pointer to the function
mov A, 0x04
mov X, 0x33
call SomeClass3Func
//Set the IDX_PP to the right page
RAM_SETPAGE_IDX >baArray
```

A similar approach can be used for calling Class 4 functions, which modify more page pointers. In this case, each page pointer register that is modified by the function must be preserved using the REG_PRESERVE and REG_RESTORE macros.

Another approach is to save all Page Pointer registers. This eliminates any risk associated with User Module code upgrades but comes at a high cost. Preserving all the Page Pointers requires a total of

80 CPU cycles and the same is true for the restoration of the registers. These requirements total 160 cycles. A macro is provided in `memory.inc` for this action.

Calling a Function while Preserving All Page Pointer Registers:

```
//Macro saves all 4 PP regs
ISR_PRESERVE_PAGE_POINTERS
//Pass a couple of vars to function
mov  A, 0x55
mov  X, 0x66
//Call the function
call  SomeClass4Func
//Restore the PP regs
ISR_RESTORE_PAGE_POINTERS
```

6.5 Interrupt Service Routines

Upon entering any interrupt service routine (ISR) the Flag register is cleared, resulting in the Page Mode bits being set to 00. Since typical ISRs are written in assembly code, even if the rest of the application is written in C, care must be taken when accessing variables within an ISR. This means that the usual rules regarding direct and index memory instructions, which apply to operation in the native mode, no longer apply. Therefore, programming the ISRs must be done with this in mind. Also, all LMM registers must not be changed during an interrupt, adding another layer of complexity in writing ISR code.

Another concern for ISRs is the process for calling API functions from within their code is properly preserving the various LMM registers. This was discussed in previous sections when looking at the different classes.

Lastly, when an interrupt occurs the Flag register must be pushed to the stack and then cleared. This means that the Page Mode bits are no longer in Native Paging Mode.

The following code illustrates how best to call an API function from within an ISR to get the proper behavior.

Some_ISR:

```
//Save the IDX_PP register
REG_PRESERVE_IDX_PP
mov  A, 0x04
//Pass a pointer to the function
mov  X, 0x33
//Restore Native Paging
RAM_RESTORE_NATIVE_PAGING
//Call the function
call  SomeClass3Func
//Pop old IDX_PP value

REG_RESTORE_IDX_PP
reti
```

The previous example shows that calling a function from within an ISR is similar to calling any other function that modifies the Page Pointer registers. The only difference is that the Native Paging must be restored before calling the function, because functions expect the program to be in its native page mode when called (see section 6.2.2, Native Paging Mode for details).

6.6 Reducing Memory Usage in LMM

The memory page architecture of Large Memory Model devices presents a programming challenge in properly setting up the page prior to executing memory instructions. Besides the complication in switching continuously between pages, a large overhead is incurred in code size and execution speed if the page management code cannot be kept at a minimum.

The best way to minimize and simplify the code is to guarantee that all variables related to each other are on the same RAM page.

The methods for doing this are listed as following:

6.6.1 Create a single memory structure and use constants to index into the structure

The linker does not allocate memory for a single variable across page boundaries. If a variable of 200 bytes is created, all the 200 bytes are guaranteed to be on the same page. An example of this is a case where there are five variables. If each variable is declared independently, the pages for these variables might not be the same, so each access to a variable requires the page pointer to be set. If a single variable is five bytes wide, the compiler places the five bytes in the same page. Constants can be used to index into the independent elements of the larger variable. When accessing these variables, the page only needs to be set for the first variable access.

An example demonstrating this concept is shown as following: Example Allocating and Accessing 5 Variables Independently:

```
AREA myArea (RAM)
Data1: blk 1
Data2: blk 1
Data3: blk 1
Data4: blk 1
Data5: blk 1
myFunction:
    RAM_SETPAGE_CUR >Data1
    mov    [Data1], A
    RAM_SETPAGE_CUR >Data2
    mov    A, [Data2]
    RAM_SETPAGE_CUR >Data3
    mov    [Data3], A
    RAM_SETPAGE_CUR >Data4
    mov    A, [Data4]
    RAM_SETPAGE_CUR >Data5
    mov    [Data5], A
```

Note that the current page pointer must be set for each variable access, because there is no guarantee that the variables reside on the same page.

Example Allocating and Accessing 5 Variables Declared as One Structure:

```

AREA myArea (RAM)
//varBlk is a variable of size 5 bytes
varBlk:blk 5
// Constants defined to access each byte of the variable
Data1     equ:0
Data2     equ:1
Data3     equ:2
Data4     equ:3
Data5     equ:4
myFunction:
    RAM_SETPAGE_CUR >varBlk
    mov [varBlk + Data1], A
    mov A, [varBlk + Data2]
    mov [varBlk + Data3], A
    mov A, [varBlk + Data4]
    mov [varBlk + Data5], A

```

Note that the current page pointer only needs to be set once at the beginning of the sequence of variable accesses.

An alternative to using structures to control the placement of variables is described in section 6.6.3.

6.6.2 Use the InterruptRAM AREA declaration to put variables on Page 0

While no guarantees can be made regarding variables in most AREA declarations, AREA InterruptRAM guarantees that all its variables are located on Page 0. Since interrupts cause the Page Mode bits to be cleared, having variables used by interrupts on Page 0 eliminates the need to switch modes and pages. The use of InterruptRAM can be extended beyond ISRs and variables that are declared in an InterruptRAM area.

The main issue with this technique is that there is only one page to use and Page 0 might already have a significant amount of space allocated for use by some of the User Modules. For Large Memory Model applications, Page 0 is no longer the stack page, which frees up additional space on Page 0. The stack is placed at the last available page by default.

InterruptRAM can be used for a set of variables frequently used in the code while fewer used variables should be placed on other pages. The following is an example of code that uses several variables to manage a large array.

```

AREA InterruptRAM(RAM, REL, CON)
    bIndex:blk 1
    bCount:blk 1
    bfStatus:blk 1

AREA myArea (RAM)
    baData:blk 256

RAM_SETPAGE_CUR >bIndex
RAM_SETPAGE_IDX >baData
rxLoop:
    movX, [bIndex]
    callGetData
    mov[X], A
    callCheckData
    tstA, BAD_DATA
    jz.skipBad

```

```

    or[bfStatus], BAD_DATA
skipBad:
    inc[bIndex]
    inc[bCount]
    jnzrxLoop

```

The previous code demonstrates that by putting the variables that are commonly used in direct memory accesses in InterruptRAM, CUR_PP needs to be set up only once. Since all indexed operations on the array (baData), IDX_PP can point to a separate page, while CUR_PP can point to page 0 where the variables that are frequently accessed reside.

6.6.3 Create unique RAM AREAs and force the linker to put each on the same page

The easiest method for placing a group of variables on a page is to create unique RAM AREAs and then modify the custom.lkp file to let the linker know that this AREA is to stay on the same page.

The following code shows how to create two unique AREAs and then how to modify the custom.lkp file to guarantee that the variables are grouped on the same page.

```

AREA func1Vars (RAM, REL, CON)
f1Var1:blk 50
f1Var2:blk 80
f1Var3: blk 100

AREA fun2Vars (RAM, REL, CON)
f2Var1:blk 50
f2Var2:blk 80
f2Var3: blk 100

```

In the custom.lkp file the following lines can be inserted to guarantee that all the data for each of these AREAs is on the same page.

-Bfunc1Vars:

-Bfunc2Vars:

The specific page used for each variable AREA can also be controlled. Some examples are as follows:

Place an Area 'BVarBlk1' on any Page:

place all variables in AREA VarBlk1 on the same page (chosen by the linker)

-BVarBlk1:

Place an Area 'BVarBlk2' on Page 2:

place all variables in AREA VarBlk2 on page 2

-BVarBlk2: 2

Place an Area 'BVarBlk3' on Page 3:

place all variables in AREA VarBlk1 on page 3

-BVarBlk3: 3

Place an Area 'BVarBlk4' on Page 4:

place all variables in AREA VarBlk1 on page 4

-BVarBlk4: 4

6.6.4 Forcing RAM Areas to Specific Pages in C

Most of you who wish to interface C and assembly languages in a single project benefit from forcing variables to particular pages. This uses the linker command "-B" in the custom.lkp file. You can create RAM areas and place them to the particular page using this linker command.

For additional details on forcing "C" variables to a specific page, refer to the C Compiler User's Guide.

6.7 LMM Macros

Along with a number of important system definitions, the memory.inc file contains a number of macros that are useful when working with the Large Memory Model. Following is a list of the important macros and their uses.

6.7.1 RAM_SETPAGE_CUR (PG_NUMBER)

Description:

Sets the Current Page Pointer (CUR_PP) to the value passed as the PG_NUMBER argument.

Usage:

```
RAM_SETPAGE_CUR A
```

```
RAM_SETPAGE_CUR >bSomeVar //This takes the page # of the variable bSomeVar
```

6.7.2 RAM_SETPAGE_STK (PG_NUMBER)

Description:

Sets the Stack Page Pointer (STK_PP) to the value passed as the PG_NUMBER argument.

Usage:

```
RAM_SETPAGE_STK A
```

```
RAM_SETPAGE_STK >bSomeVar //This takes the page # of the variable bSomeVar
```

6.7.3 RAM_SETPAGE_IDX (PG_NUMBER)

Description:

Sets the Index Page Pointer (IDX_PP) to the value passed as the PG_NUMBER argument.

Usage:

```
RAM_SETPAGE_IDX A
```

```
RAM_SETPAGE_IDX >bSomeVar //This takes the page # of the variable bSomeVar
```

6.7.4 6.7.4RAM_SETPAGE_MVR (PG_NUMBER)

Description:

Sets the MVI Read Page Pointer (MVR_PP) to the value passed as the PG_NUMBER argument.

Usage:

```
RAM_SETPAGE_MVR A
```

```
RAM_SETPAGE_MVR >bSomeVar //This takes the page # of the variable bSomeVar
```

6.7.5 RAM_SETPAGE_MVW (PG_NUMBER)

Description:

Sets the MVI Write Page Pointer (MVW_PP) to the value passed as the PG_NUMBER argument.

Usage:

```
RAM_SETPAGE_MVW A
RAM_SETPAGE_MVW >bSomeVar //This takes the page # of the variable bSomeVar
```

6.7.6 RAM_SETPAGE_IDX2STK

Description:

Sets the value of IDX_PP to the Stack Page Constant defined in *memory.inc*. This is useful in order to perform Indexed Mode Memory instructions on the stack page while in Page Mode 2.

Usage:

```
RAM_SETPAGE_IDX2STK
```

6.7.7 RAM_CHANGE_PAGE_MODE (MODE)

Description:

Forces the Page Mode bits to the value passed as the MODE argument.

Usage:

```
RAM_CHANGE_PAGE_MODE FLAG_PGMODE_2
```

6.7.8 RAM_RESTORE_NATIVE_PAGING

Description:

Uses the constants defined in *memory.inc* to force a return to the Native Paging Mode defined in the *project.mk* or *local.mk* file

Usage:

```
RAM_RESTORE_NATIVE_PAGING
```

6.7.9 RAM_SET_NATIVE_PAGING

Description:

This is a faster way of returning to native paging but it assumes that the Page Mode bits have been cleared.

Usage:

```
RAM_SET_NATIVE_PAGING
```

6.7.10 RAM_X_POINTS_TO_STACKPAGE

Description:

Changes the Page Mode bits in such a way as to guarantee that Indexed Mode Memory instructions operate on whatever page is specified by the STK_PP register.

Usage:

```
RAM_X_POINTS_TO_STACKPAGE
```

6.7.11 RAM_X_POINTS_TO_INDEXPAGE

Description:

Assuming that the Page Mode bits are in a Native Page Mode, this macro changes the Page Mode bits in such a way as to guarantee that the Indexed Mode Memory instructions operate on whatever page is specified by the IDX_PP register.

Usage:

```
RAM_X_POINTS_TO_INDEXPAGE
```

6.7.12 REG_PRESERVE (IOReg)

Description:

Pushes the register specified by the IOReg variable onto the stack. This macro modifies A before pushing to the stack.

Usage:

```
REG_PRESERVE CUR_PP
```

```
REG_PRESERVE IDX_PP
```

6.7.13 REG_PRESERVE (IOReg)

Description:

Pops the value at the top of the stack and places it in the register specified by the IOReg variable. This macro modifies A.

Usage:

```
REG_RESTORE CUR_PP
```

```
REG_RESTORE IDX_PP
```

6.7.14 ISR_PRESERVE_PAGE_POINTERS

Description:

Pushes all the Page Pointer registers, with the exception of STK_PP, onto the stack. This macro modifies A.

Usage:

```
ISR_PRESERVE_PAGE_POINTERS
```

6.7.15 ISR_RESTORE_PAGE_POINTERS

Description:

Pops all the top 4 values on the stack and sequentially writes back to the Page Pointer registers to mirror the process by which they were pushed if ISR_PRESERVE_PAGE_POINTERS was used. This macro modifies A.

Usage:

```
ISR_RESTORE_PAGE_POINTER
```


7. Builds and Error Messages



This chapter briefly describes the PSoC Designer assemble and build process, linker operations, and errors you might encounter with your code.

7.1 Assemble and Build

Once you have added and modified assembly language source files, you must assemble the files and build the project. This is done so PSoC Designer can generate a HEX file to be used to download to the ICE and debug the PSoC program. Each time you assemble files or build the project, the Output Status window is cleared and the current status is entered as the process occurs.



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.

When building is complete, you will see the number of errors. Zero errors signifies that the assemblage or build was successful. One or more errors indicate problems with one or more files. For more information on the PSoC Designer Output Status Window refer to the *PSoC Designer IDE Guide*.

7.2 Linker Operations

The main purpose of the Linker is to combine multiple object files into a single output file, suitable to be downloaded to the In-Circuit Emulator for debugging the code and programming the device. Linking takes place in PSoC Designer when a project build is executed. The linker can also take input from a library which is basically a file containing multiple object files. In producing the output file, the Linker resolves any references between the input files. In some detail, the linking steps involve:

1. Making the startup file (*boot.asm*) the first file to be linked. The startup file initializes the execution environment for the C program to run.
2. Appending any libraries that you explicitly request (or in most cases, as are requested by the IDE) to the list of files to be linked. Library modules that are directly or indirectly referenced will be linked. All user-specified object files (e.g., your program files) are linked.
3. Scanning the object files to find unresolved references. The linker marks the object file (possibly in the library) that satisfies the references and adds it to its list of unresolved references. It repeats the process until there are no outstanding unresolved references.
4. Combining all marked object files into an output file, and generating map and listing files as needed.

For additional information about the Linker and specifying Linker settings, refer to the *PSoC Designer IDE Guide*.

7.3 Code Compressor and Dead-Code Elimination Error Messages

Problem –

!X The compiler has failed an internal consistency check. This may be due to incorrect input or an internal error. Please report the information target == 0 || new_target at ..\optm8c.c(340) to "Cypress" at www.cypress.com/support.

```
Designer\tools\make: *** [output/drc_test.rom] Error 1
```

Note To obtain support go to <http://www.cypress.com/support/login.cfm> or www.cypress.com and click on Technical and Support KnowledgeBase at the bottom of the page.

Possible Causes –

1. The label in a `.LITERAL` or `.SECTION` segment of code has not been made global using the `EXPORT` directive or a double colon.
2. A `.LITERAL` segment has only a label and no defined data.
 - a. `.SECTION` was not followed by a label.
 - b. `.LITERAL` was not followed by a label.
 - c. `.ENDSECTION` has no matching `.SECTION`.
 - d. `.ENDLITERAL` has no matching `.LITERAL`.
 - e. `.SECTION` has no `.ENDSECTION`.
 - f. Unmatched `.LITERAL` directive.
 - g. Directive creating data may not be compatible with Code Compression and other advanced technologies.
3. Data defined in ROM does not have the `.LITERAL` and `.ENDLITERAL` directives.

A. Reference Tables Appendix



The tables in this appendix are intended to serve as a quick reference to the M8C assembler directives. The tables are also found in the body of this guide. For detailed information on the instruction set and the assembler directives, refer to the [Instruction Set Summary on page 14](#) and the [Assembler Directives chapter on page 75](#).

A.1 Assembly Syntax Expressions

Table A-1. Assembly Syntax Expressions

Precedence	Expression	Symbol	Form
1	Bitwise Complement	~	(~a)
2	Multiplication/Division/Modulo	*, /, %	(a*b), (a/b), (a%b)
3	Addition / Subtraction	+, -	(a+b), (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)

A.2 Operand Constant Formats.

Table A-2. Constants Formats

Radix	Name	Formats	Example
127	ASCII Character	'J'	mov A, 'J' ;character constant mov A, '\\' ;use "\" to escape "\" mov A, '\\\\' ;use "\" to escape "\"
16	Hexadecimal	0x4A 4Ah \$4A	mov A, 0x4A ;hex--"0x" prefix mov A, 4Ah ;hex--append "h" mov A, \$4A ;hex--"\$" prefix
10	Decimal	74	mov A, 74 ;decimal--no prefix
8	Octal	0112	mov A, 0112 ;octal--zero prefix
2	Binary	0b01001010 %01001010	mov A, 0b01001010 ;bin--"0b" prefix mov A, %01001010 ;bin--"%" prefix

A.3 Assembler Directives Summary

Table A-3. Assembler Directives Summary

Symbol	Directive
AREA	Area
ASCIZ	NULL Terminated ASCII String
BLK	RAM Byte Block
BLKW	RAM Word Block
DB	Define Byte
DS	Define ASCII String
DSU	Define UNICODE String
DW	Define Word
DWL	Define Word With Little Endian Ordering
ELSE	Alternative Result of IF Directive
ENDIF	End Conditional Assembly
ENDM	End Macro
EQU	Equate Label to Variable Value
EXPORT	Export
IF	Start Conditional Assembly
INCLUDE	Include Source File
.LITERAL, .ENDLITERAL	Prevent Code Compression of Data
MACRO	Start Macro Definition
ORG	Area Origin
.SECTION, .ENDSECTION	Section for Dead-Code Elimination
Suspend - OR F,0 Resume - ADD SP,0	Suspend and Resume Code Compressor

A.4 ASCII Code Table

Table A-4. ASCII Code Table

Dec	HEX	Oct	Char	Dec	HEX	Oct	Char	Dec	HEX	Oct	Char	Dec	HEX	Oct	Char
0	00	000	NULL	32	20	040	space	64	40	100	@	96	60	140	'
1	01	001	SOH	33	21	041	!	65	41	101	A	97	61	141	a
2	02	002	STX	34	22	042	"	66	42	102	B	98	62	142	b
3	03	003	ETX	35	23	043	#	67	43	103	C	99	63	143	c
4	04	004	EOT	36	24	044	\$	68	44	104	D	100	64	144	d
5	05	005	ENQ	37	25	045	%	69	45	105	E	101	65	145	e
6	06	006	ACK	38	26	046	&	70	46	106	F	102	66	146	f
7	07	007	BEL	39	27	047	'	71	47	107	G	103	67	147	g
8	08	010	BS	40	28	050	(72	48	110	H	104	68	150	h
9	09	011	HT	41	29	051)	73	49	111	I	105	69	151	i
10	0A	012	LF	42	2A	052	*	74	4A	112	J	106	6A	152	j
11	0B	013	VT	43	2B	053	+	75	4B	113	K	107	6B	153	k
12	0C	014	FF	44	2C	054	,	76	4C	114	L	108	6C	154	l
13	0D	015	CR	45	2D	055	-	77	4D	115	M	109	6D	155	m
14	0E	016	SO	46	2E	056	.	78	4E	116	N	110	6E	156	n
15	0F	017	SI	47	2F	057	/	79	4F	117	O	111	6F	157	o
16	10	020	DLE	48	30	060	0	80	50	120	P	112	70	160	p
17	11	021	DC1	49	31	061	1	81	51	121	Q	113	71	161	q
18	12	022	DC2	50	32	062	2	82	52	122	R	114	72	162	r
19	13	023	DC3	51	33	063	3	83	53	123	S	115	73	163	s
20	14	024	DC4	52	34	064	4	84	54	124	T	116	74	164	t
21	15	025	NAK	53	35	065	5	85	55	125	U	117	75	165	u
22	16	026	SYN	54	36	066	6	86	56	126	V	118	76	166	v
23	17	027	ETB	55	37	067	7	87	57	127	W	119	77	167	w
24	18	030	CAN	56	38	070	8	88	58	130	X	120	78	170	x
25	19	031	EM	57	39	071	9	89	59	131	Y	121	79	171	y
26	1A	032	SUB	58	3A	072	:	90	5A	132	Z	122	7A	172	z
27	1B	033	ESC	59	3B	073	;	91	5B	133	[123	7B	173	{
28	1C	034	FS	60	3C	074	<	92	5C	134	\	124	7C	174	
29	1D	035	GS	61	3D	075	=	93	5D	135]	125	7D	175	}
30	1E	036	RS	62	3E	076	>	94	5E	136	^	126	7E	176	~
31	1F	037	US	63	3F	077	?	95	5F	137	_	127	7F	177	DEL

A.5 Instruction Set Summary

Table A-5. Instruction Set Summary Sorted Numerically by Opcode

Opcode HEX	Cycles	Bytes	Instruction Format	Flags	Opcode HEX	Cycles	Bytes	Instruction Format	Flags	Opcode HEX	Cycles	Bytes	Instruction Format	Flags
00	15	1	SSC		2D	8	2	OR [X+expr], A	Z	5A	5	2	MOV [expr], X	
01	4	2	ADD A, expr	C, Z	2E	9	3	OR [expr], expr	Z	5B	4	1	MOV A, X	Z
02	6	2	ADD A, [expr]	C, Z	2F	10	3	OR [X+expr], expr	Z	5C	4	1	MOV X, A	
03	7	2	ADD A, [X+expr]	C, Z	30	9	1	HALT		5D	6	2	MOV A, reg[expr]	Z
04	7	2	ADD [expr], A	C, Z	31	4	2	XOR A, expr	Z	5E	7	2	MOV A, reg[X+expr]	Z
05	8	2	ADD [X+expr], A	C, Z	32	6	2	XOR A, [expr]	Z	5F	10	3	MOV [expr], [expr]	
06	9	3	ADD [expr], expr	C, Z	33	7	2	XOR A, [X+expr]	Z	60	5	2	MOV reg[expr], A	
07	10	3	ADD [X+expr], expr	C, Z	34	7	2	XOR [expr], A	Z	61	6	2	MOV reg[X+expr], A	
08	4	1	PUSH A		35	8	2	XOR [X+expr], A	Z	62	8	3	MOV reg[expr], expr	
09	4	2	ADC A, expr	C, Z	36	9	3	XOR [expr], expr	Z	63	9	3	MOV reg[X+expr], expr	
0A	6	2	ADC A, [expr]	C, Z	37	10	3	XOR [X+expr], expr	Z	64	4	1	ASL A	C, Z
0B	7	2	ADC A, [X+expr]	C, Z	38	5	2	ADD SP, expr		65	7	2	ASL [expr]	C, Z
0C	7	2	ADC [expr], A	C, Z	39	5	2	CMP A, expr		66	8	2	ASL [X+expr]	C, Z
0D	8	2	ADC [X+expr], A	C, Z	3A	7	2	CMP A, [expr]	if (A=B) Z=1 if (A<B) C=1	67	4	1	ASR A	C, Z
0E	9	3	ADC [expr], expr	C, Z	3B	8	2	CMP A, [X+expr]		68	7	2	ASR [expr]	C, Z
0F	10	3	ADC [X+expr], expr	C, Z	3C	8	3	CMP [expr], expr		69	8	2	ASR [X+expr]	C, Z
10	4	1	PUSH X		3D	9	3	CMP [X+expr], expr		6A	4	1	RLC A	C, Z
11	4	2	SUB A, expr	C, Z	3E	10	2	MVI A, [[expr]++]	Z	6B	7	2	RLC [expr]	C, Z
12	6	2	SUB A, [expr]	C, Z	3F	10	2	MVI [[expr]++], A		6C	8	2	RLC [X+expr]	C, Z
13	7	2	SUB A, [X+expr]	C, Z	40	4	1	NOP		6D	4	1	RRC A	C, Z
14	7	2	SUB [expr], A	C, Z	41	9	3	AND reg[expr], expr	Z	6E	7	2	RRC [expr]	C, Z
15	8	2	SUB [X+expr], A	C, Z	42	10	3	AND reg[X+expr], expr	Z	6F	8	2	RRC [X+expr]	C, Z
16	9	3	SUB [expr], expr	C, Z	43	9	3	OR reg[expr], expr	Z	70	4	2	AND F, expr	C, Z
17	10	3	SUB [X+expr], expr	C, Z	44	10	3	OR reg[X+expr], expr	Z	71	4	2	OR F, expr	C, Z
18	5	1	POP A	Z	45	9	3	XOR reg[expr], expr	Z	72	4	2	XOR F, expr	C, Z
19	4	2	SBB A, expr	C, Z	46	10	3	XOR reg[X+expr], expr	Z	73	4	1	CPL A	Z
1A	6	2	SBB A, [expr]	C, Z	47	8	3	TST [expr], expr	Z	74	4	1	INC A	C, Z
1B	7	2	SBB A, [X+expr]	C, Z	48	9	3	TST [X+expr], expr	Z	75	4	1	INC X	C, Z
1C	7	2	SBB [expr], A	C, Z	49	9	3	TST reg[expr], expr	Z	76	7	2	INC [expr]	C, Z
1D	8	2	SBB [X+expr], A	C, Z	4A	10	3	TST reg[X+expr], expr	Z	77	8	2	INC [X+expr]	C, Z
1E	9	3	SBB [expr], expr	C, Z	4B	5	1	SWAP A, X	Z	78	4	1	DEC A	C, Z
1F	10	3	SBB [X+expr], expr	C, Z	4C	7	2	SWAP A, [expr]	Z	79	4	1	DEC X	C, Z
20	5	1	POP X		4D	7	2	SWAP X, [expr]		7A	7	2	DEC [expr]	C, Z
21	4	2	AND A, expr	Z	4E	5	1	SWAP A, SP	Z	7B	8	2	DEC [X+expr]	C, Z
22	6	2	AND A, [expr]	Z	4F	4	1	MOV X, SP		7C	13	3	LCALL	
23	7	2	AND A, [X+expr]	Z	50	4	2	MOV A, expr	Z	7D	7	3	LJMP	
24	7	2	AND [expr], A	Z	51	5	2	MOV A, [expr]	Z	7E	10	1	RETI	C, Z
25	8	2	AND [X+expr], A	Z	52	6	2	MOV A, [X+expr]	Z	7F	8	1	RET	
26	9	3	AND [expr], expr	Z	53	5	2	MOV [expr], A		8x	5	2	JMP	
27	10	3	AND [X+expr], expr	Z	54	6	2	MOV [X+expr], A		9x	11	2	CALL	
28	11	1	ROMX	Z	55	8	3	MOV [expr], expr		Ax	5	2	JZ	
29	4	2	OR A, expr	Z	56	9	3	MOV [X+expr], expr		Bx	5	2	JNZ	
2A	6	2	OR A, [expr]	Z	57	4	2	MOV X, expr		Cx	5	2	JC	
2B	7	2	OR A, [X+expr]	Z	58	6	2	MOV X, [expr]		Dx	5	2	JNC	
2C	7	2	OR [expr], A	Z	59	7	2	MOV X, [X+expr]		Ex	7	2	JACC	
										Fx	13	2	INDEX	Z

Note 1 Interrupt acknowledge to Interrupt Vector table = 13 cycles.

Note 2 The number of cycles required by an instruction is increased by one for instructions that span 256 byte page boundaries in the Flash memory space.

Table A-6. Instruction Set Summary Sorted Alphabetically by Mnemonic

Opcode HEX	Cycles	Bytes	Instruction Format	Flags	Opcode HEX	Cycles	Bytes	Instruction Format	Flags	Opcode HEX	Cycles	Bytes	Instruction Format	Flags
09	4	2	ADC A, expr	C, Z	76	7	2	INC [expr]	C, Z	20	5	1	POP X	
0A	6	2	ADC A, [expr]	C, Z	77	8	2	INC [X+expr]	C, Z	18	5	1	POP A	Z
0B	7	2	ADC A, [X+expr]	C, Z	Fx	13	2	INDEX	Z	10	4	1	PUSH X	
0C	7	2	ADC [expr], A	C, Z	Ex	7	2	JACC		08	4	1	PUSH A	
0D	8	2	ADC [X+expr], A	C, Z	Cx	5	2	JC		7E	10	1	RETI	C, Z
0E	9	3	ADC [expr], expr	C, Z	8x	5	2	JMP		7F	8	1	RET	
0F	10	3	ADC [X+expr], expr	C, Z	Dx	5	2	JNC		6A	4	1	RLC A	C, Z
01	4	2	ADD A, expr	C, Z	Bx	5	2	JNZ		6B	7	2	RLC [expr]	C, Z
02	6	2	ADD A, [expr]	C, Z	Ax	5	2	JZ		6C	8	2	RLC [X+expr]	C, Z
03	7	2	ADD A, [X+expr]	C, Z	7C	13	3	LCALL		28	11	1	ROMX	Z
04	7	2	ADD [expr], A	C, Z	7D	7	3	LJMP		6D	4	1	RRC A	C, Z
05	8	2	ADD [X+expr], A	C, Z	4F	4	1	MOV X, SP		6E	7	2	RRC [expr]	C, Z
06	9	3	ADD [expr], expr	C, Z	50	4	2	MOV A, expr	Z	6F	8	2	RRC [X+expr]	C, Z
07	10	3	ADD [X+expr], expr	C, Z	51	5	2	MOV A, [expr]	Z	19	4	2	SBB A, expr	C, Z
38	5	2	ADD SP, expr		52	6	2	MOV A, [X+expr]	Z	1A	6	2	SBB A, [expr]	C, Z
21	4	2	AND A, expr	Z	53	5	2	MOV [expr], A		1B	7	2	SBB A, [X+expr]	C, Z
22	6	2	AND A, [expr]	Z	54	6	2	MOV [X+expr], A		1C	7	2	SBB [expr], A	C, Z
23	7	2	AND A, [X+expr]	Z	55	8	3	MOV [expr], expr		1D	8	2	SBB [X+expr], A	C, Z
24	7	2	AND [expr], A	Z	56	9	3	MOV [X+expr], expr		1E	9	3	SBB [expr], expr	C, Z
25	8	2	AND [X+expr], A	Z	57	4	2	MOV X, expr		1F	10	3	SBB [X+expr], expr	C, Z
26	9	3	AND [expr], expr	Z	58	6	2	MOV X, [expr]		00	15	1	SSC	
27	10	3	AND [X+expr], expr	Z	59	7	2	MOV X, [X+expr]		11	4	2	SUB A, expr	C, Z
70	4	2	AND F, expr	C, Z	5A	5	2	MOV [expr], X		12	6	2	SUB A, [expr]	C, Z
41	9	3	AND reg[expr], expr	Z	5B	4	1	MOV A, X	Z	13	7	2	SUB A, [X+expr]	C, Z
42	10	3	AND reg[X+expr], expr	Z	5C	4	1	MOV X, A		14	7	2	SUB [expr], A	C, Z
64	4	1	ASL A	C, Z	5D	6	2	MOV A, reg[expr]	Z	15	8	2	SUB [X+expr], A	C, Z
65	7	2	ASL [expr]	C, Z	5E	7	2	MOV A, reg[X+expr]	Z	16	9	3	SUB [expr], expr	C, Z
66	8	2	ASL [X+expr]	C, Z	5F	10	3	MOV [expr], [expr]		17	10	3	SUB [X+expr], expr	C, Z
67	4	1	ASR A	C, Z	60	5	2	MOV reg[expr], A		4B	5	1	SWAP A, X	Z
68	7	2	ASR [expr]	C, Z	61	6	2	MOV reg[X+expr], A		4C	7	2	SWAP A, [expr]	Z
69	8	2	ASR [X+expr]	C, Z	62	8	3	MOV reg[expr], expr		4D	7	2	SWAP X, [expr]	
9x	11	2	CALL		63	9	3	MOV reg[X+expr], expr		4E	5	1	SWAP A, SP	Z
39	5	2	CMP A, expr	if (A=B) Z=1 if (A<B) C=1	3E	10	2	MVI A, [[expr]++]	Z	47	8	3	TST [expr], expr	Z
3A	7	2	CMP A, [expr]		3F	10	2	MVI [[expr]++], A		48	9	3	TST [X+expr], expr	Z
3B	8	2	CMP A, [X+expr]		40	4	1	NOP		49	9	3	TST reg[expr], expr	Z
3C	8	3	CMP [expr], expr		29	4	2	OR A, expr	Z	4A	10	3	TST reg[X+expr], expr	Z
3D	9	3	CMP [X+expr], expr		2A	6	2	OR A, [expr]	Z	72	4	2	XOR F, expr	C, Z
73	4	1	CPL A	Z	2B	7	2	OR A, [X+expr]	Z	31	4	2	XOR A, expr	Z
74	4	1	DEC A	C, Z	2C	7	2	OR [expr], A	Z	32	6	2	XOR A, [expr]	Z
79	4	1	DEC X	C, Z	2D	8	2	OR [X+expr], A	Z	33	7	2	XOR A, [X+expr]	Z
7A	7	2	DEC [expr]	C, Z	2E	9	3	OR [expr], expr	Z	34	7	2	XOR [expr], A	Z
7B	8	2	DEC [X+expr]	C, Z	2F	10	3	OR [X+expr], expr	Z	35	8	2	XOR [X+expr], A	Z
30	9	1	HALT		43	9	3	OR reg[expr], expr	Z	36	9	3	XOR [expr], expr	Z
74	4	1	INC A	C, Z	44	10	3	OR reg[X+expr], expr	Z	37	10	3	XOR [X+expr], expr	Z
75	4	1	INC X	C, Z	71	4	2	OR F, expr	C, Z	45	9	3	XOR reg[expr], expr	Z
										46	10	3	XOR reg[X+expr], expr	Z

Note 1 Interrupt acknowledge to Interrupt Vector table = 13 cycles.

Note 2 The number of cycles required by an instruction is increased by one for instructions that span 256 byte page boundaries in the Flash memory space.

Index



A

- absolute table read instruction 67
- ADD instruction 39
- ADD SP,0 directive 94
- add with carry instruction 38
- add without carry instruction 39
- address spaces 12
- addressing modes, M8C 18
- AND instruction 40
- AREA directive 76
- area origin directive 93
- arithmetic shift left instruction 41
- arithmetic shift right instruction 42
- ASCII code table 121
- ASCIZ directive 78
- ASL instruction 41
- ASR instruction 42
- assembler
 - comments 29
 - directives 30, 75
 - errors and warnings 117
 - Intel HEX file format 32
 - labels 26
 - listing file format 30
 - map file format 30
 - mnemonics 27
 - operands 28
 - ROM file format 30
 - source file format 25
- assembly syntax expressions 119

B

- bitwise AND instruction 40
- bitwise OR instruction 61
- bitwise XOR instruction 74
- BLK directive 79
- BLKW directive 80
- build current project 117

C

- call function instruction 43
- CALL instruction 43
- CMP instruction 44
- compiling file into library module 34
- compiling source files 117

- complement accumulator instruction 45
- components of assembly source file 25
- compressor and dead code error message
 - elimination 102, 118
- conditional source directive 89
- constants format table 119
- conventions 8
- CPL instruction 45
- CPU core
 - addressing modes 18
 - instruction formats 16
 - instruction set summary 14–15, 122

D

- DB directive 81
- debugging 99, 117
- DEC instruction 46
- decrement instruction 46
- define ASCII string directive 83
- define byte directive 81
- define floating-point number directive 82
- define UNICODE string directive 84
- define word, big endian ordering directive 85
- define word, little endian ordering directive 86
- destination instructions
 - direct 20
 - direct source direct 22
 - direct source immediate 21
 - indexed 20
 - indexed source immediate 21
 - indirect post increment 23
- DF directive 82
- directives summary 75, 120
- documentation
 - conventions 8
 - overview 7
- DS directive 83
- DSU directive 84
- DW directive 85
- DWL directive 86

E

- elimination of compressor and dead code error
 - messages 102, 118
- ELSE directive 89

ENDIF directive 89
ENDLITERAL directive 91
ENDM directive 92
ENDSECTION directive 94
EQU directive 87
equate label directive 87
errors 117

G

global labels 27

H

HALT instruction 47
help, getting 8, 117

I

IF directive 89
INC instruction 48
INCLUDE directive 90
include source file directive 90
increment instruction 48
INDEX instruction 49
instruction formats
 1-byte instructions 16
 2-byte instructions 16
 3-byte instructions 17
instruction set summary 14–15, 122
instruction set, M8C 37
Intel HEX file format 32
internal registers
 accumulator 11
 flags 11
 index 11
 program counter 11
 restoring 34
 stack pointer 11
introduction 7

J

JACC instruction 50
JC instruction 51
JMP instruction 52
JNC instruction 53
JNZ instruction 54
jump accumulator instruction 50
jump if carry 51
jump if no carry instruction 53
jump if not zero instruction 54
jump if zero instruction 55
jump instruction 52
JZ instruction 55

L

LCALL instruction 56
library module, compiling file 34
linker
 operations 99, 117
listing file format 30
LITERAL directive 91
LJMP instruction 57
local labels 26
long call instruction 56
long jump instruction 57

M

M8C microprocessor 11
 address spaces 12
 addressing modes 18
 instruction formats 16
 instruction set 37
 instructions set summary 14
 internal registers 11
macro definition directive 92
MACRO directive 92
map file format 30
mnemonics 27
MOV instruction 58
move indirect, post-increment to memory
 instruction 59
move instruction 58
MVI instruction 59

N

no operation instruction 60
non-destructive compare instruction 44
NOP instruction 60
NULL terminated ASCII string directive 78

O

operands
 constants 28
 constants format table 119
 dot operator 28
 expressions 29
 labels 28
 RAM 29
 registers 29
OR F,0 directive 94
OR instruction 61
ORG directive 93
overview of chapters 7

P

POP instruction 62
pop stack into register instruction 62

prevent code compression of data 91
product
 support 8
 upgrades 8
PUSH instruction 63
push register onto stack instruction 63

R

RAM block in bytes directive 79
RAM block in words directive 80
relative table read instruction 49
restoring internal registers 34
resume code compressor directive 94
RET instruction 64
RETI instruction 65
return from interrupt instruction 65
return instruction 64
re-usable local labels 27
RLC instruction 66
ROM file format 30
ROMX instruction 67
rotate left through carry instruction 66
rotate right through carry instruction 68
RRC instruction 68

S

SBB instruction 69
SECTION directive 94
section for dead-code elimination directive 94
source file components
 comments 29
 directives 30
 labels 26
 mnemonics 27
 operands 28
source file format 25
source instructions
 direct 19
 immediate 18
 indexed 19
 indirect post increment 22
SSC instruction 72
SUB instruction 70
subtract with borrow instruction 69
subtract without borrow instruction 70
support 8
suspend code compressor directive 94
SWAP instruction 71
syntax expressions 119
system supervisor call instruction 72

T

technical support 8
test for mask instruction 73

TST instruction 73

U

upgrades 8

W

warnings 117

X

XOR instruction 74

