

Chapter 2

Microcontroller Architecture

The chapter presents the main characteristics of a microcontroller instruction set, and discusses programming techniques in assembly language for several applications. It also defines the instruction set architecture of PSoC's M8C microcontroller.

The instruction set of the M8C microcontroller consists of instructions for (i) data transfer, (ii) arithmetic operations, (iii) logic operations, (iv) execution flow control, and (v) other miscellaneous instructions.

The microcontroller provides ten addressing modes, which result from combining four basic modes: immediate, direct, indexed, and indirect addressing. Each addressing mode defines a specific tradeoff among the execution time of the code, the required program memory, and the flexibility in modifying the code.

The PSoC memory space includes SRAM for storing data, and nonvolatile memory for storing the program code and the predefined subroutines used in booting up the architecture, accessing the flash memory that holds the application program, and circuit calibration. In addition, the register space stores the status and control information of the embedded mixed-signal architecture.

Several applications illustrate programming techniques in assembly language, and discuss the performance of these solutions. The applications include data block transfer, stack operation, unsigned data multiplication, calling assembly routines from programs in high-level programming languages, bit manipulations, and sequence detectors.

The chapter has the following structure:

- Section 1 presents M8C's instruction set and addressing modes.
- Section 2 explains PSoC's SRAM and ROM subsystems.
- Section 3 presents chapter conclusions.

2.1 Microcontroller Architecture

PSoC's M8C microcontroller is based on an eight-bit Harvard architecture with separate data and address buses. Figure 2.1 shows the architecture of the M8C, that is the microcontroller structure, its internal registers and external memory space [2].

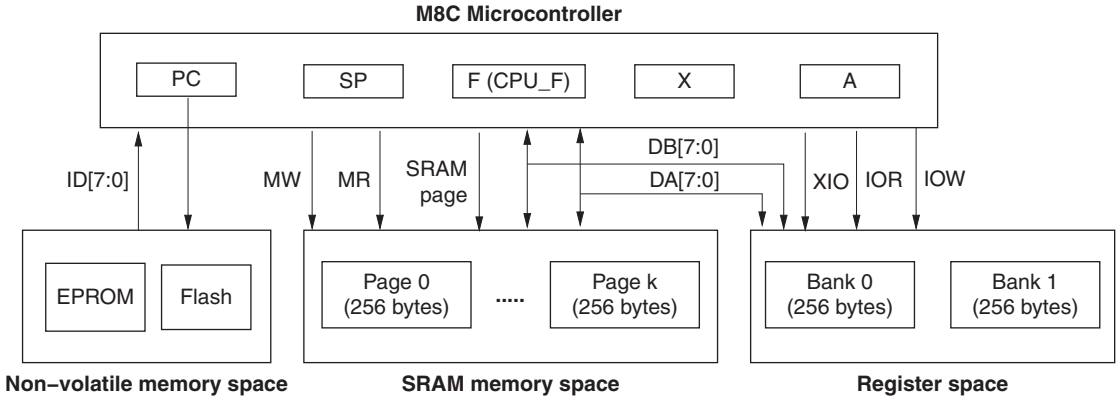


Figure 2.1: The M8C microcontroller structure [2].

The M8C has five internal registers.

- A Register (Accumulator) an eight bit, general-purpose register used by instructions that involve data transfer, arithmetic and logical operations, jump instructions, and so on.
- X Register (Index Register) an eight bit register that can be used either as a general-purpose register, similar to register A, or for implementing certain addressing modes, for example source-indexed and destination indexed addressing.
- F Register (Flag Register), also referred to as the CPU_F register an eight bit, nonaddressable register located at address x,F7H.¹ This register stores the various flag and control bits of the microcontroller utilizing the following bit structure.
 - Bit 0 is the GIE (Global Interrupt Enable) bit and determines which of the external interrupts are enabled, or disabled.
 - Bit 1 is the ZF (Zero Flag) bit, and Bit 2, the CF bit (Carry Flag). Both bits are set by certain types of data transfers, or by data processing instructions.
 - Bit 4 is the XIO bit (IO Bank Select), determines the active register bank.
 - Bits 6 and 7 are the PgMode bits (Page Mode). They control the accessing of data stored in SRAM. More details are offered in Subsection 2.2.
- SP Register (Stack Pointer) an eight bit register that points to the top of the stack. The SRAM page of the stack is pointed to by the eight-bit STD_PP register, in the register space.
- PC Register (Program Counter) stores the 16-bit program memory address, representing 64K of memory space, and points to the next instruction to be executed.

The PSoC memory space consists of three distinct memory regions:

- *Nonvolatile memory space* consisting of the permanent read-only memory (EPROM) and flash memory, that are used to store the program code to be executed by the M8C processor. The size of the flash memory space can be up to 64K words, inclusive. The address of a memory word is pointed to by the PC register. The data read is returned via a dedicated eight-bit bus, called ID[7:0], as shown in Figure 2.1.

¹An 'x' before the comma in the address field indicates that this register can be read or written to, no matter what bank is used.

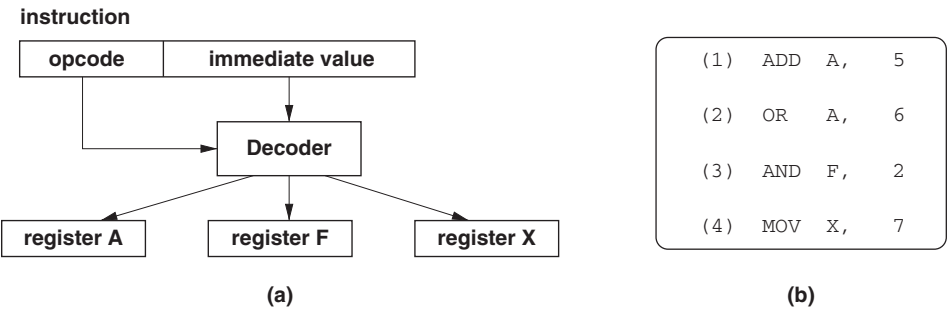


Figure 2.2: Source-immediate addressing mode.

- *SRAM space* stores both global and local variables, and the stack implementation. The maximum size of the SRAM space is 2048 words, inclusive, and the number of pages is limited to eight. Because each page is 256 bytes long, eight-bit address words are required to access an SRAM page. The accessing of pages is controlled by the “control bits” in the CPU_F (CPU Flags) and CUR_PP (Current Page Pointer) registers. The SRAM and register spaces share the same eight-bit address and data buses, DA and DB, respectively. Control signals MR and MW indicate a memory read and a memory write, respectively.
- *Register space* consists of the registers needed to control PSoC’s resources, for example the digital and analog reconfigurable blocks, SRAM and interrupt system. Additional details about the individual registers are provided as PSoC’s programmable hardware is discussed later in the text.

The M8C architecture includes two register banks that are selected by bit four (bit XIO) of the CPU_F register. Having two register banks is useful for *dynamic hardware reconfiguration* (i.e., when the hardware is reconfigured during execution, to provide different functionality and/or performance). Each of the banks stores the control information for a configuration mode.

(Subsection 2.2 provides additional details about the PSoC memory system.)

2.1.1 Microcontroller Addressing Modes

The microcontroller addressing modes impose different conventions, that is rules for generating the address used in accessing SRAM. Address modes range from very simple address generation rules, such as utilizing the physical address of the word, specified in a field of the instruction word, to the implementation of more complex rules that use index registers, or indirect addressing based on pointers.

Different addressing modes impose different tradeoffs with respect to the (i) flexibility in addressing memory, (ii) number of instructions required to prepare a memory address, (iii) number of clock cycles required to access memory, and (iv) the number of registers utilized by an addressing mode. Simple addressing modes, for example those specifying the operand value, or indicating the physical address as part of the instruction word, offer faster memory access, execute in fewer clock cycles, and require fewer instructions to generate the address. The major disadvantage of such addressing schemes is their inflexibility, because they impose a rigid mapping of data to memory words. Unless consistent addressing guidelines are followed, and reused in each new application, it is difficult to link modules with each other.

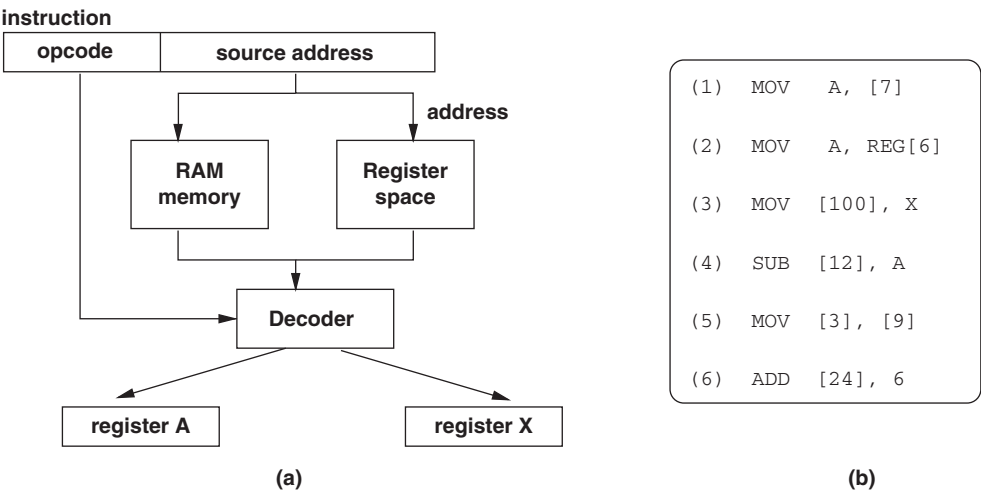


Figure 2.3: Source-direct and destination-direct addressing modes.

The M8C microcontroller supports the following addressing modes.

A. Source-Immediate Addressing

Figure 2.2(a) illustrates source-immediate addressing. Instructions using this mode include a field that contains the value of one of the instruction operands. The other operand, and the result, are kept in registers A, X, and F, respectively.

Figure 2.2(b) shows four instructions that use source-immediate addressing. The first instruction adds the value 5 to the value in register A, and the result is stored in register A. The second instruction performs a bitwise OR operation on the bits of register A and the mask ‘00000110’. As a result, bits 1 and 2 of register A are set to the value ‘1’ and the rest of the bits in register A are unchanged. The third instruction describes a bitwise AND operation between register F, the register holding the flag and control bits, and the mask ‘00000010’. This instruction resets all bits of register F except for bit 1, which is left unchanged. The fourth instruction loads the value “7” into register X.

B. Source-Direct and Destination-Direct Addressing

Figure 2.3(a) describes source-direct and destination-direct addressing. The source address field of the instruction, for source-direct addressing, contains the address of a location in either the SRAM space, or the register space. For example, the first instruction in Figure 2.3(b), MOV A, [7], points to the SRAM cell at the physical address 7. The value found at this address is loaded into register A. The second instruction is MOV A, REG[6] and refers to the register at address 6 of the register space. Its contents are loaded into the A register.

Destination-direct addressing uses a field of the instruction word to store the destination address. A destination location can be in SRAM or in the register space. The third instruction in Figure 2.3(b) uses destination-direct addressing. The value in the X register is loaded in SRAM

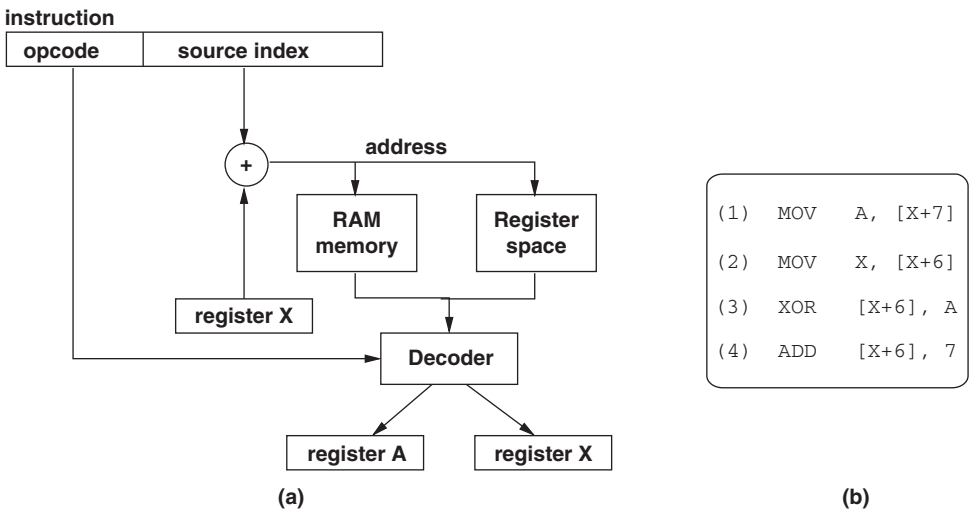


Figure 2.4: Source-indexed and destination-indexed addressing modes.

at the physical address 100. Similarly, the fourth instruction subtracts the value in register A from the value at SRAM address 12, and the result is stored at SRAM address 12.

Source-direct and destination-direct addressing are combined in instructions that include two address fields, one to specify the address of the source and the other, the destination address. This combined mode is called destination-direct source-direct addressing [2]. Instruction five, in Figure 2.3(b), illustrates a “move” instruction using source-direct and destination-direct addressing. In this example, the value in memory address 9 is copied into location with the memory address 3.

Source-immediate and destination-direct addressing can also be employed simultaneously by an instruction. One field of the instruction stores the value of one of the source operands, using source-immediate addressing. A second field contains the address of the destination, which for certain instructions also represents the destination-direct address of the second operand. Instruction six, in Figure 2.3(b), illustrates the addition instruction in which the source operand is accessed by immediate addressing (the operand’s value is 6), the second operand is at the SRAM address 24, and the result is stored at address 24. Destination-direct addressing is used to determine the destination.

C. Source-Indexed and Destination-Indexed Addressing

Figure 2.4(a) is a summary of the characteristics of source-indexed and destination-indexed addressing. For source-indexed addressing, the instruction includes a field for the source index that represents the relative displacement of the addressed memory location with respect to a base address stored in the index register X. Hence, the address of the accessed memory location is obtained by adding the content of the X register and the source index field. The address can point to a location in either the SRAM space, or to the register space.

The address of the memory location is loaded into the A or X register, depending on the value of the instruction opcode. Figure 2.4(b) shows two instructions using source-indexed addressing.

`MOV A,[X+7]` loads the A register with the value of the memory cell at address register X + 7.
`MOV X,[X+6]` loads the X register with the value at address register X + 6.

Instructions using destination-indexed addressing include a field that specifies the offset of the destination related to the reference address stored in the X register. Instruction three in Figure 2.4 shows an exclusive OR (XOR) instruction with one operand located in the A register and the second operand at the address given by the value “register X + 6”. The result of the instruction is stored at the address of the second operand, that is “register X + 6”.

Destination-indexed and source-immediate addressing can also be combined. These instructions use one of their fields to define a value, for example, used as an operand in an arithmetic operation, and a second field to store the offset of the second operand and result, as in the case of destination-indexed addressing. The fourth instruction in Figure 2.4(b) shows an addition instruction using destination-indexed and source-immediate addressing. One operand has a value of 7. The second operand is found at the address given by the result of “register X + 6”. The result is stored at the same address.

D. Source-Indirect and Destination-Indirect Postincrement Addressing

Instructions using source indirect postincrement addressing are often used in transferring blocks of data, and include a field containing the SRAM address of a pointer to the source data. After executing the instruction, the value of the pointer is incremented, so that the next datum in the block can be accessed. The pointer is always located in the current memory page pointed to by the CUR_PP register at the address in the register space. This pointer references data located in the SRAM page pointed to by the MVR_PP register. (Additional details are given in the sections on MVI instructions, the only instructions that use this mode.)

Destination-indirect postincrement addressing uses a similar addressing scheme for the destination. The instruction field contains the address of a pointer used to refer to the destination. After executing this instruction, the value of the pointer is incremented. The pointer is always located in the current SRAM page pointed to by the CUR_PP register. The destination is located in the SRAM page pointed to by the MVW_PP register. MVI instructions are the only ones that use this addressing mode. (Additional details are given in the sections on MVI instructions.)

The mechanism for these two addressing modes is summarized in Figure 2.5.

2.1.2 Instruction Set

The M8C instruction set includes instructions for (i) data transfer, (ii) arithmetic operations, (iii) logical operations, (iv) execution flow control, and (v) other miscellaneous instructions. These instructions use source-immediate, source-direct, source-indexed, source-indirect-postincrement, destination-direct, destination-indexed, and destination-postincrement addressing.

A. Instructions for Data Transfer

Instructions for data transfer include the following instructions: MOV, MVI, SWAP, POP, PUSH, ROMX, and INDEX instructions. The instructions are detailed next.

A.1 MOV and MVI instructions

MOV instructions transfer data among the A, SP, and X registers, SRAM space, and register space. Table 2.1 lists the different kinds of *MOV* instructions. For each instruction, the table

Table 2.1: **MOV** instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
MOV X,SP	$X \leftarrow SP$	0x4F	1	4
MOV A,expr	$A \leftarrow \text{expr}$	0x50	2	4
MOV A,[expr]	$A \leftarrow \text{SRAM}[\text{expr}]$	0x51	2	5
MOV A,[X+expr]	$A \leftarrow \text{SRAM}[X+\text{expr}]$	0x52	2	6
MOV [expr],A	$\text{SRAM}[\text{expr}] \leftarrow A$	0x53	2	5
MOV [X+expr],A	$\text{SRAM}[X+\text{expr}] \leftarrow A$	0x54	2	6
MOV [expr₁],expr₂	$\text{SRAM}[\text{expr}_1] \leftarrow \text{expr}_2$	0x55	3	8
MOV [X+expr₁],expr₂	$\text{SRAM}[X+\text{expr}_1] \leftarrow \text{expr}_2$	0x56	3	9
MOV X,expr	$X \leftarrow \text{expr}$	0x57	2	4
MOV X,[expr]	$X \leftarrow \text{SRAM}[\text{expr}]$	0x58	2	6
MOV X,[X+expr]	$X \leftarrow \text{SRAM}[X+\text{expr}]$	0x59	2	7
MOV [expr],X	$\text{SRAM}[\text{expr}] \leftarrow X$	0x5A	2	5
MOV A,X	$A \leftarrow X$	0x5B	1	4
MOV X,A	$X \leftarrow A$	0x5C	1	4
MOV A,REG[expr]	$A \leftarrow \text{REG}[\text{expr}]$	0x5D	2	6
MOV A,REG[X+expr]	$A \leftarrow \text{REG}[X+\text{expr}]$	0x5E	2	7
MOV [expr₁],[expr₂]	$\text{SRAM}[\text{expr}_1] \leftarrow \text{SRAM}[\text{expr}_2]$	0x5F	3	10
MOV REG[expr],A	$\text{REG}[\text{expr}] \leftarrow A$	0x60	2	5
MOV REG[X+expr],A	$\text{REG}[X+\text{expr}] \leftarrow A$	0x61	2	6
MOV REG[expr₁],expr₂	$\text{REG}[\text{expr}_1] \leftarrow \text{expr}_2$	0x62	3	8
MOV REG[X+expr₁],expr₂	$\text{REG}[X+\text{expr}_1] \leftarrow \text{expr}_2$	0x63	3	9

shows the opcode of the instruction, the number of bytes occupied by the instruction, and the number of clock cycles required to execute the instruction.

The semantics of the instructions and the addressing mode to be used are as follows.

- *MOV X,SP*: The value of the SP register is loaded into the X register.
- *MOV A,expr*: The value of expr is loaded into the A register. This instruction uses immediate addressing for accessing the source value to be loaded into the register.
- *MOV A,[expr]*: The value found at address expr in SRAM is loaded into the A register (source-direct addressing).
- *MOV A,[X+expr]*: The value in the SRAM, at the address given by the value “X register + expr”, is loaded into the A register (source-indexed addressing).
- *MOV [expr],A*: This instruction copies the value in the A register to the SRAM cell at address expr (destination-direct addressing).
- *MOV [X+expr],A*: The value in the A register is stored in the SRAM at the address “X register + expr” (indexed addressing for the destination).
- *MOV [expr₁],expr₂*: Value expr₂ is loaded into SRAM at address expr₁ (source-immediate and destination-direct addressing).
- *MOV [X+expr₁],expr₂*: Value expr₂ is stored in SRAM at address “X register + expr₁” (source-immediate and destination indexed addressing).
- *MOV X,expr*: The X register is loaded with the value expr (source-immediate addressing).
- *MOV X,[expr]*: The value in SRAM, at address expr, is copied into the X register (source-direct addressing).
- *MOV X,[X+expr]*: The X register is loaded with the value found in SRAM at address “X register + expr” (source-indexed addressing).
- *MOV [expr],X*: The content of the X register is stored in SRAM at the address expr (destination-direct addressing).
- *MOV A,X*: The A register is loaded with the value in the X register.
- *MOV X, A*: The value in the A register is copied into the X register.
- *MOV A,REG[expr]*: This instruction loads the A register with the value in register expr, which is in the register space.
- *MOV A,REG[X+expr]*: Using the source-indexed addressing mode, the A register is loaded with the value in the register at address “X register + expr”.
- *MOV REG[expr], A*: Register expr is loaded with the value of the A register.
- *MOV REG[X+expr],A* - The value in the A register is copied into the register pointed to by the value of “X register + expr” (destination indexed addressing).
- *MOV REG[expr₁],expr₂* - Value expr₂ is loaded into the register selected by expr₁ (destination-direct addressing).
- *MOV REG[X+expr₁],expr₂* - Value expr₂ is stored into the register pointed by expression “X register + expr₁” (destination indexed addressing).

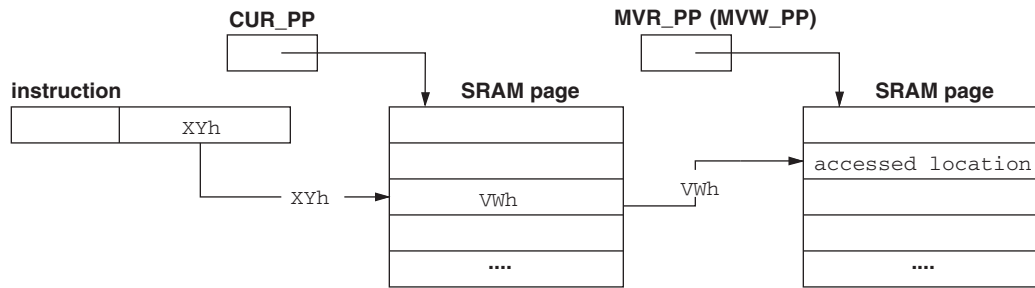


Figure 2.5: MVI instructions.

If the value zero is loaded into the A register, the ZF flag (Zero Flag) is set and the CF flag (Carry Flag) remains unchanged.

MVI instructions implement data transfers using source indirect and destination indirect addressing. Table 2.2 shows the two types of MVI instructions. In contrast to MOV instructions, the expr field of a MVI instruction is a pointer to a SRAM cell, for example the contents of SRAM at address expr is an address of another SRAM cell.

- *MVI A,[expr]* loads the A register with the contents of the memory cell pointed to by [expr], and [expr] is incremented. The SRAM cell is in the SRAM page pointed to by the MVR_PP register. The last three bits of the MVR_PP (MVR_PP[2:0]) register at address 0,D4H select the SRAM page for MVI instructions (source indirect addressing).
- *MVI [expr],A* stores the value found in the A register in the SRAM cell referred to by the pointer at address expr and the pointer is incremented. The SRAM cell is located in the SRAM page pointed to by the MVW_PP register. The last three bits of the MVW_PP register, that is bits 2-0, (MVW_PP[2:0]), at address 0,D5H, select the SRAM page for the MVI instruction (destination indirect addressing).

The CF flag is unchanged by these instructions. If the value zero is loaded into register A, the flag ZF is set. Figure 2.5 illustrates the source indirect post-increment and destination indirect post-increment addressing used by MVI instructions. The address field of the instruction points to a location in the current SRAM page selected by the CUR_PP register. In the figure, this location is pointed to by XYh. Then, the content of the selected location, shown as value VWh in the figure, is used as a pointer to a memory location in the SRAM page selected by the MVR_PP register for MVI A, [expr] instructions, and by the MVW_PP register for MVI [expr],A instructions. This is the memory location used in the data transfer.

Table 2.2: MVI instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
MVI A,[expr]	$A \leftarrow \text{SRAM}[\text{SRAM}[\text{expr}]]$ $\text{SRAM}[\text{expr}] \leftarrow \text{SRAM}[\text{expr}] + 1$	0x3E	2	10
MVI [expr], A	$\text{SRAM}[\text{SRAM}[\text{expr}]] \leftarrow A$ $\text{SRAM}[\text{expr}] \leftarrow \text{SRAM}[\text{expr}] + 1$	0x3F	2	10

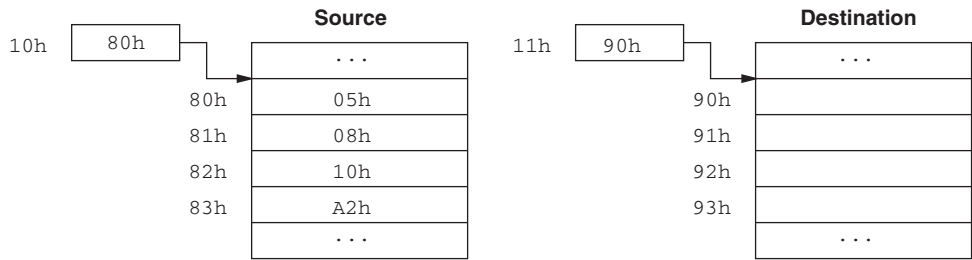


Figure 2.6: Data vector transfer example.

Characteristics of MOV and MVI instructions

As shown in Table 2.1, there is a variety of ways in which data can be transferred in a system. Immediate, direct, indexed, and indirect post-increment addressing have specific execution times that are expressed in terms of clock cycles. Each requires different amounts of memory for storing their code, expressed in bytes, and some modes involve the A and X registers. Finally, the four addressing modes offer specific flexibility and benefits in terms of reusable, assembly language, source code that can be incorporated into new applications.

With respect to execution times, accessing the A register requires the shortest time, followed by X register accesses, register space accesses, and finally SRAM accesses, which are the slowest. Immediate addressing is the fastest, followed by direct addressing, and indexed addressing. MOV [expr₁],[expr₂] requires the longest execution time, viz., ten clock cycles, because both source and destination are located in SRAM, and involve direct addressing.

Example (Execution time for addressing modes). The following example illustrates the performance characteristics of the four addressing modes. The goal is to transfer a data vector of size 4 from address A in the SRAM to address B in memory. In this example, a data value is one byte long, and the four data values are stored at consecutive memory locations. Figure 2.6 is a graphical representation of the operations to be performed. Table 2.3 shows the performance characteristics of the four description styles, namely, the execution time in clock cycles, nonvolatile (flash) memory requirement in bytes, corresponding number of instructions, and the flexibility in the development of new programs.

If the four values to be transferred are known in advance, immediate addressing can be used for data transfer. Figure 2.7(a) shows the source code for this example. Each of the MOV instructions uses source-immediate addressing to point to the value to be transferred, and destination-direct addressing to point directly to the location that the data are to be copied to in memory. The data transfer is fastest in this case, requiring only 32 clock cycles, which is approximately three times less than the number of clock cycles required for the slowest case, that is using indirect post-increment addressing. Also, the flash memory requirement is the smallest of the four cases, with only 12 bytes of memory being required to store the four instructions. However, this method provides little flexibility, which can be a significant disadvantage. If other values have to be transferred, or if data are stored at different memory locations, then the code has to be modified.

Figure 2.7(b) shows the assembly language source code required for the data transfer if source and destination-direct addressing are used. The flash memory requirements are still small (only

(1) MOV [90h], 05h	(1) MOV [90h], [80h]
(2) MOV [91h], 08h	(2) MOV [91h], [81h]
(3) MOV [92h], 10h	(3) MOV [92h], [82h]
(4) MOV [93h], A2h	(4) MOV [93h], [83h]
(a)	(b)
(1) MOV X, 00h	(1) MOV [10h], 80h
(2) MOV A, [X+80h]	(2) MOV [11h], 90h
(3) MOV [X+90h], A	(3) MVI A, [10h]
(4) INC X	(4) MVI [11h], A
(5) MOV A, [X+80h]	(5) MVI A, [10h]
(6) MOV [X+90h], A	(6) MVI [11h], A
(7) INC X	(7) MVI A, [10h]
(8) MOV A, [X+80h]	(8) MVI [11h], A
(9) MOV [X+90h], A	(9) MVI A, [10h]
(10) INC X	(10) MVI [11h], A
(11) MOV A, [X+80h]	
(12) MOV [X+90h], A	
(13) INC X	
(c)	(d)

Figure 2.7: Assembly code for data vector transfer.

12 bytes), but the execution time is slightly longer than for the first case. Forty clock cycles are required for the data transfer and although the flexibility is somewhat better, overall it is still poor. Although this code is independent of the data values that are transferred, it will have to be changed if different memory addresses and regions for source and destination are involved in the transfer.

Figure 2.7(c) shows the source code using indexed addressing. The X register stores the index of the data involved in the transfer. This register is initialized to zero, and incremented after each transfer. Because of the instructions available, a data value has to be copied from the source location to the A register, and then moved from the A register to the destination location. This requires thirteen instructions, which occupy 22 bytes of flash memory, almost double the amount needed for the first two cases. Although the execution time is also much longer (viz., 68 clock cycles) which is more than twice the time for the case in Figure 2.7(a), this case affords greater flexibility. If different memory regions are involved in the transfer, then only the base addresses of the regions (i.e., 80h for the source, and 90h for the destination) have to be updated. This is easily achieved by modifying the associated label values for the source and destination.

Figure 2.7(d) shows the source code, using indirect post-increment addressing. The memory locations at addresses 10h and 11h store the base addresses for the source and destination regions. Using MVI instructions, a data value is loaded from the source region into the A register, and then moved to the destination region. No increment instructions are needed, as in the third case, because the pointer values are automatically incremented. As a result only ten instructions are required, which occupy 22 bytes in memory, as in the indexed addressing case. The execution

time is 96 clock cycles, the longest time of the four cases. The flexibility is similar to the third case except that the labels for the two base addresses have to be changed if different memory regions are involved in the data transfer.

Clearly, if flexibility and the ease of changing the source code is the main concern, indexed addressing should be used. If minimal execution time is the focus, then direct addressing provides fast data transfer while retaining flexibility.

In general, the execution time, in terms of clock cycles required for transferring a data vector of size K from address A to address B , can be expressed by:

$$\text{Number of clock cycles} = K \times \sum_{i=1}^n \text{cycles}_i \tag{2.1}$$

where n is the number of instructions required to transfer one data value, i is the index for the set of instructions required for the transfer of one data value and cycles_i represents the number of cycles required for the i -th instruction.

The execution time depends on the number of data items, K , to be transferred and the number of clock cycles required to transfer each. To reduce the execution time, direct addressing should be used when small, fixed-size amounts of data need to be accessed, and indexed addressing should be employed if large datasets, or sets of unknown size, are involved. In addition, the relative positioning of the allocated memory for the source and target also influences the number of clock cycles required. For example, manipulating compact memory regions requires the execution of fewer instructions and hence fewer clock cycles than if data are not in contiguous memory regions.

Table 2.3: Characteristics of the four specification styles.

	Immediate	Direct	Indexed	Indirect
Execution time (clock cycles)	32	40	68	96
flash memory (number of bytes)	12	12	22	22
Number of instructions	4	4	13	10
Flexibility	very low	low	high	high

A.2 SWAP Instructions

Swap instructions exchange the contents of the A, X, SP registers and SRAM, respectively. Table 2.4 lists the four kinds of SWAP instructions and is a summary of the semantics of each instruction, its opcode, the number of bytes occupied by the instruction in ROM, and the number of clock cycles required for its execution.

The semantics of the SWAP instructions is as follows:

- *SWAP A,X*: This instruction swaps the contents of the A and X registers.
- *SWAP A,[expr]*: The contents of the A register is swapped with the contents of the SRAM cell at address *expr* (source-direct addressing mode).
- *SWAP X,[expr]*: This instruction swaps the contents of the X register with the contents of the SRAM cell at address *expr* (source-direct addressing).
- *SWAP A,SP*: The contents of the A and SP registers are exchanged.

While SWAP instructions do not modify the CF flag, the ZF flag is set if a value of zero is loaded into the A register.

Table 2.4: **SWAP** instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
SWAP A, X	$\text{aux} \leftarrow \text{X}; \text{X} \leftarrow \text{A}$ $\text{A} \leftarrow \text{aux}$	0x4B	1	5
SWAP A, [expr]	$\text{aux} \leftarrow \text{SRAM}[\text{expr}]; \text{SRAM}[\text{expr}] \leftarrow \text{A}$ $\text{A} \leftarrow \text{aux}$	0x4C	2	7
SWAP X, [expr]	$\text{aux} \leftarrow \text{SRAM}[\text{expr}]; \text{SRAM}[\text{expr}] \leftarrow \text{X}$ $\text{X} \leftarrow \text{aux}$	0x4D	2	7
SWAP A, SP	$\text{aux} \leftarrow \text{SP}; \text{SP} \leftarrow \text{A}$ $\text{A} \leftarrow \text{aux}$	0x4E	1	5

A.3 POP and PUSH Instructions

POP and PUSH instructions are used to implement stack operations. PUSH places a value on the stack, at the location pointed to by the stack pointer, and then increments the stack pointer register. POP returns a value from the stack, and decrements the stack pointer register. Table 2.5 lists the various types of POP and PUSH instructions. This table is a summary of the semantics of the instructions, their respective opcodes, the number of bytes they occupy in SRAM, and the number of clock cycles required for execution.

The semantics for POP and PUSH instructions is as follows:

- *POP A*: The SRAM cell value at address $\text{SP} - 1$ is loaded into the A register. The stack pointer, stored in the SP register, is decremented.
- *POP X*: The SRAM cell value at address $\text{SP} - 1$ is loaded into the X register. The stack pointer is decremented.
- *PUSH A* - The content of the A register is stored on the stack at the location pointed to by SP register. The stack pointer is incremented.
- *PUSH X* - The value in register X is stored on the stack. The stack pointer is incremented.

Table 2.5: **POP** and **PUSH** instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
POP A	$A \leftarrow \text{SRAM}[\text{SP}-1]$ $\text{SP} \leftarrow \text{SP}-1$	0x18	1	5
POP X	$X \leftarrow \text{SRAM}[\text{SP}-1]$ $\text{SP} \leftarrow \text{SP}-1$	0x20	1	5
PUSH A	$\text{SRAM}[\text{SP}] \leftarrow A$ $\text{SP} \leftarrow \text{SP}+1$	0x08	1	4
PUSH X	$\text{SRAM}[\text{SP}] \leftarrow X$ $\text{SP} \leftarrow \text{SP}+1$	0x10	1	4

POP and PUSH instructions do not modify the CF flag and PUSH instructions do not affect the ZF flag. If a value of zero is loaded by a POP instruction into the Aregister, the ZF flag is set.

The SRAM page, which can be located in any of the eight SRAM pages, is selected by the three least significant bits of the STD_PP (STD_PP[2:0]) register located at address 0,D1H. Because the stack is located in a single SRAM page, after reaching address FFH, the stack pointer, SP, returns to address 00H. Upon reset, the default value of the STD_PP register is set to 00H, and stack operations then refer to SRAM page zero, unless the STD_PP register is modified. Figure 2.8(a) shows the stack implementation for the M8C.

Example. Figures 2.8(b) and (c) show the source code for “safe POP” and “safe PUSH” instructions. The safe PUSH operation places a value on the stack only if the stack is not full. Similarly, the safe POP operation pops a value from the stack only if the stack is not empty. Location 60h in SRAM stores the number of values on the stack and the value at location 61h indicates the size of the stack. If the number of values is equal to the stack size, then the “zero flag” is set after the SUB instruction in line 2 of the Figure 2.8(b). In this case, execution jumps to the label END_PUSH. Otherwise, if the stack is not full, the value is pushed onto the stack (instruction 5) and the variable holding the number of values is incremented (instruction 6). The pushed value is found in the SRAM at address 80h (instruction 4).

The safe POP operation first checks whether there are values in the stack. If the number of values is zero (i.e., the stack is empty) and the ZF flag is set after executing instruction 1 in Figure 2.8(c) execution jumps to the “end” label. Otherwise, the value on top of the stack is first POP-ped into the A register (instruction 3), and then copied into SRAM at address 81h (instruction 4). Finally, the variable defining the number of values in the stack is decremented (instruction 5).

A.4 Data transfer from nonvolatile memory: ROMX and INDEX instructions

ROMX instructions transfer data, for ceample global constants, from the nonvolatile memory (i.e. flash memory) to the A register. Table 2.6 is a summary of the semantics of the instruction, its opcode, the number of bytes required to store the instruction, and the number of clock cycles required for execution. The current value of the PC register is stored in two temporary locations, t_1 and t_2 with the less significant byte in the former and the more significant byte in the latter. The address of the memory location to be accessed is determined by the contents of the X and A registers. The X register contains the less significant byte of the address, and the A register the more significant byte. After loading the PC register with the address, the value is copied into the A register. Finally, the PC register is restored to the value saved in the temporary locations, t_1 and t_2 .

Table 2.6: ROMX instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
ROMX	$t_1 \leftarrow PC[7:0]; PC[7:0] \leftarrow X;$ $t_2 \leftarrow PC[15:8]; PC[15:8] \leftarrow A;$ $A \leftarrow ROM[PC]; PC[7:0] \leftarrow t_1;$ $PC[15:8] \leftarrow t_2$	0x28	1	11

If the A register is loaded with a value of zero, then the zero flag is set. However, the Carry Flag is not changed.

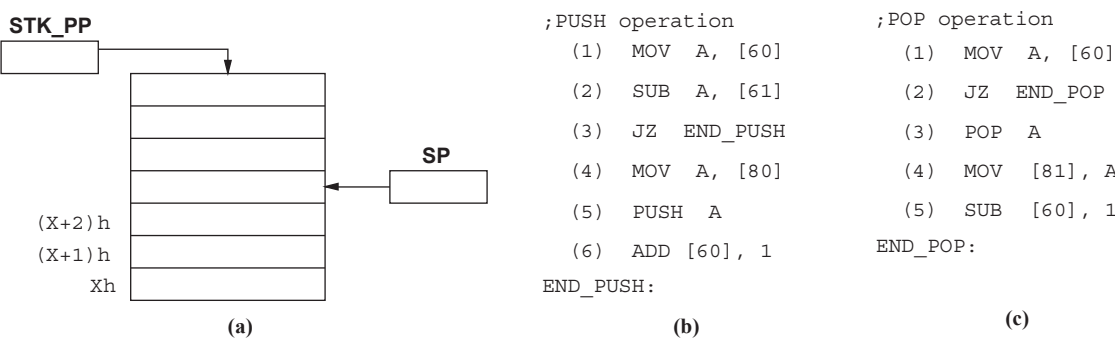


Table 2.7: **INDEX** instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
INDEX <i>expr</i>	$A \leftarrow \text{ROM}[A + \text{expr} + \text{PC}]$ $-2048 \leq \text{expr} \leq 2047$	0xFx	2	13

INDEX instructions are useful for transferring data stored in tables in the flash memory. The A register is loaded with the contents of the flash memory address obtained by adding the values stored in A, *expr*, and PC. Table 2.7 is a summary of the main attributes of this instruction. The index value that points to a table entry is computed as shown in Figure 2.9. *expr* is a 12-bit offset relative to the program counter, and is expressed as its two's complement. Hence, its most significant bit is the sign bit. The remaining 11 bits represent offsets in the range -2048 to 2047. The sum of PC and *expr* gives the starting address of the table and the A register contains the table offset of the entry to be accessed.

Using INDEX instructions to access tables is illustrated by an example at the end of this chapter.

B. Instructions for Arithmetic Operations

Instructions for arithmetic operations include the following instructions: ADC, ADD, SBB, SUB, INC, DEC, CMP, ASL, ASR, RLC, and RRC instructions. The instructions are presented next.

B.1 Instruction for Addition: ADC and ADD Instructions

ADC and ADD instructions perform addition operations. The ADC instructions add the value of the CF flag and the two operands. The ADD instructions add the operands but do not add the CF flag. Table 2.8 presents the different kinds of ADC and ADD instructions. This table is a summary of the semantics of the instructions, and shows their opcode, the number of bytes required to store the instruction in the memory, and the number of clock cycles required for execution.

The semantics of ADC and ADD instructions is as follows:

- *ADC A,expr*: This instruction adds the contents of the A register, the value of expression *expr*, and the CF flag. The result is stored in the A register (source-immediate addressing).
- *ADC A,[expr]*: The contents of the A register are increased by the value found at address *expr* in SRAM plus the CF flag (source-direct addressing).
- *ADC A,[X+expr]*: This instruction adds the contents of the A register, the value in SRAM at address “X register + *expr*”, and the CF flag, and stores the result into the A register (source-indexed addressing).
- *ADC [expr],A*: The SRAM contents at address *expr* are added to the contents of the A register and the CF flag, and the result is stored in SRAM at address *expr* (destination-direct addressing).

Table 2.8: ADC and ADD instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
ADC A,expr	$A \leftarrow A + \text{expr} + \text{CF}$	0x09	2	4
ADC A,[expr]	$A \leftarrow A + \text{SRAM}[\text{expr}] + \text{CF}$	0x0A	2	6
ADC A,[X+expr]	$A \leftarrow A + \text{SRAM}[X+\text{expr}] + \text{CF}$	0x0B	2	7
ADC [expr],A	$\text{SRAM}[\text{expr}] \leftarrow \text{SRAM}[\text{expr}] + A + \text{CF}$	0x0C	2	7
ADC [X+expr],A	$\text{SRAM}[X+\text{expr}] \leftarrow \text{SRAM}[X+\text{expr}] + A + \text{CF}$	0x0D	2	8
ADC [expr ₁],expr ₂	$\text{SRAM}[\text{expr}_1] \leftarrow \text{SRAM}[\text{expr}_1] + \text{expr}_2 + \text{CF}$	0x0E	3	9
ADC [X+expr ₁],expr ₂	$\text{SRAM}[X+\text{expr}_1] \leftarrow \text{SRAM}[X+\text{expr}_1] + \text{expr}_2 + \text{CF}$	0x0F	3	10
ADD A,expr	$A \leftarrow A + \text{expr}$	0x01	2	4
ADD A,[expr]	$A \leftarrow A + \text{SRAM}[\text{expr}]$	0x02	2	6
ADD A,[X+expr]	$A \leftarrow A + \text{SRAM}[X+\text{expr}]$	0x03	2	7
ADD [expr],A	$\text{SRAM}[\text{expr}] \leftarrow \text{SRAM}[\text{expr}] + A$	0x04	2	7
ADD [X+expr],A	$\text{SRAM}[X+\text{expr}] \leftarrow \text{SRAM}[X+\text{expr}] + A$	0x05	2	8
ADD [expr ₁],expr ₂	$\text{SRAM}[\text{expr}_1] \leftarrow \text{SRAM}[\text{expr}_1] + \text{expr}_2$	0x06	3	9
ADD [X+expr ₁],expr ₂	$\text{SRAM}[X+\text{expr}_1] \leftarrow \text{SRAM}[X+\text{expr}_1] + \text{expr}_2$	0x07	3	10
ADD SP,expr	$\text{SP} \leftarrow \text{SP} + \text{expr}$	0x38	2	5

- *ADC [X+expr],A*: The SRAM value at address “X + expr” is added to the value in the A register and the CF flag, and the result is stored in SRAM at address register “X + expr” (destination indexed addressing).
- *ADC [expr₁],expr₂*: The SRAM contents at address expr₁ are added to the value of expr₂ and the CF flag, and the result is saved at SRAM address expr₁. This instruction uses immediate addressing for referring to the source, and direct addressing for referring to the destination.
- *ADC [X+expr₁],expr₂*: This instruction adds the value of expr₂ to the value found in SRAM at address “X + expr₁” and the CF flag, and stores the result at the latter address (source-immediate and destination indexed addressing).
- *ADD A,expr*: The value of the A register is increased by the value of expr (source-immediate addressing).
- *ADD A,[expr]*: The contents of the A register are added to the value found in the SRAM at address expr, and the result is stored in the A register (source-direct addressing).
- *ADD A,[X+expr]*: The value in the A register is increased by the memory value pointed to by X register indexed with the value expr (source-indexed addressing).
- *ADD [expr],A*: The SRAM value at address expr is increased by the value in the A register (destination-direct addressing).
- *ADD [X+expr],A*: The memory value at address “X register + expr” is increased by the value stored in the A register (destination indexed addressing).
- *ADD [expr₁],expr₂*: The memory contents at address expr₁ are increased by the value expr₂.
- *ADD [X+expr₁],expr₂* - Value expr₂ is added to the value in memory at address “register X + expr₁”, and the result is stored at the same address.
- *ADD SP,expr* - The contents of the SP register are increased by the value of expr.

ADC and ADD instructions set the ZF flag if the result is zero, and clear the flag otherwise. If the result is larger than 255, i.e., the maximum value that can be represented by eight bits, the CF flag is set to one. If not, CF is set to zero.

B.2 Instruction for Subtraction: SBB and SUB Instructions

SBB and *SUB* instructions execute subtraction operations. The *SBB* instructions subtract the value of the CF flag and the two operands. The *SUB* instructions subtract the operands but not the CF flag. Table 2.9 presents the different kinds of *SBB* and *SUB* instructions. This table is a summary of the semantics of the instructions, and shows their opcode, the number of bytes required to store the instruction in the memory, and the number of clock cycles needed for execution.

The semantics of *SBB* and *SUB* instructions is as follows.

- *SBB A,expr*: subtracts the values of expr and CF flag from the A register. The result is stored in the A register (source-immediate addressing).
- *SBB A,[expr]* - The value found at address expr in SRAM plus the CF flag are subtracted from the A register (source-direct addressing).

Table 2.9: *SBB* and *SUB* instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
SBB A, expr	$A \leftarrow A - (\text{expr} + \text{CF})$	0x19	2	4
SBB A, [expr]	$A \leftarrow A - (\text{SRAM}[\text{expr}] + \text{CF})$	0x1A	2	6
SBB A, [X+expr]	$A \leftarrow A - (\text{SRAM}[X+\text{expr}] + \text{CF})$	0x1B	2	7
SBB [expr], A	$\text{SRAM}[\text{expr}] \leftarrow \text{SRAM}[\text{expr}] - (A + \text{CF})$	0x1C	2	7
SBB [X+expr], A	$\text{SRAM}[X+\text{expr}] \leftarrow \text{SRAM}[X+\text{expr}] - (A + \text{CF})$	0x1D	2	8
SBB [expr₁], expr₂	$\text{SRAM}[\text{expr}_1] \leftarrow \text{SRAM}[\text{expr}_1] - (\text{expr}_2 + \text{CF})$	0x1E	3	9
SBB [X+expr₁], expr₂	$\text{SRAM}[X+\text{expr}_1] \leftarrow \text{SRAM}[X+\text{expr}_1] - (\text{expr}_2 + \text{CF})$	0x1F	3	10
SUB A, expr	$A \leftarrow A - \text{expr}$	0x11	2	4
SUB A, [expr]	$A \leftarrow A - \text{SRAM}[\text{expr}]$	0x12	2	6
SUB A, [X+expr]	$A \leftarrow A - \text{SRAM}[X+\text{expr}]$	0x13	2	7
SUB [expr], A	$\text{SRAM}[\text{expr}] \leftarrow \text{SRAM}[\text{expr}] - A$	0x14	2	7
SUB [X+expr], A	$\text{SRAM}[X+\text{expr}] \leftarrow \text{SRAM}[X+\text{expr}] - A$	0x15	2	8
SUB [expr₁], expr₂	$\text{SRAM}[\text{expr}_1] \leftarrow \text{SRAM}[\text{expr}_1] - \text{expr}_2$	0x16	3	9
SUB [X+expr₁], expr₂	$\text{SRAM}[X+\text{expr}_1] \leftarrow \text{SRAM}[X+\text{expr}_1] - \text{expr}_2$	0x17	3	10

- *SBB A,[X+expr]*: - The value at address “register X + expr” in memory and the Carry Flag are subtracted from the A register. The result is stored in the A register (source-indexed addressing).
- *SBB [expr],A*: - The A register and the CF flag are subtracted from the value in SRAM at address expr (destination-direct addressing).
- *SBB [X+expr],A*: Similar to the previous instruction, the A register and CF flag values are subtracted from the SRAM value at address “register X + expr” (destination indexed addressing).
- *SBB [expr₁],expr₂*: This instruction subtracts the value expr₂ and the CF flag from the value at address expr₁ in the memory. The result is stored in the memory at address expr₁ (source-immediate addressing and destination-direct addressing).
- *SBB [X+expr₁],expr₂*: The memory contents at the address pointed to by the X register, indexed with the value expr₁ is decreased by the value expr₂ plus the CF flag (source-immediate and destination indexed addressing),
- *SUB A,expr*: The value represented by expr is subtracted from the A register, and the result is stored in the A register (source-immediate addressing).
- *SUB A,[expr]*: The value found at address expr in SRAM is subtracted from the A register, and the result is stored in the A register (source-direct addressing).
- *SUB A,[X+expr]*: The value in the A register is decreased by the value of memory entry at the address pointed to by the X register plus the value expr (source-indexed addressing).
- *SUB [expr],A*: subtracts the contents of the A register from the value at address expr in the memory, and the result is stored in memory at the same address (destination-direct addressing mode).
- *SUB [X+expr],A*: The A register is subtracted from the memory value at address “register X + expr”, and the result is stored in memory at the same address (destination indexed addressing).
- *SUB [expr₁],expr₂*: This instruction subtracts the value expr₂ from the value at address expr₁ in memory. The result is stored in the memory at address expr₁ (source-immediate addressing and destination-direct addressing).
- *SUB [X+expr₁],expr₂*: This instruction subtracts the value expr₂ from the value at address given by “register X + expr₁”: (source-immediate and destination indexed addressing).

B.3 Instruction for Incrementing and Decrementing: INC and DEC Instructions

The *INC* instructions increment the values in the A and X registers, and SRAM. Table 2.10 lists the four kinds of increment instructions and shows the semantics of the instructions, their opcodes, the number of memory bytes, and their respective execution times.

The *DEC* instructions decrement the values in the A and X registers, and SRAM. Table 2.11 lists the four kinds of decrement instructions and shows the semantics of the instructions, their opcodes, the number of memory bytes, and their respective execution times.

Table 2.10: **INC** instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
INC A	$A \leftarrow A + 1$	0x74	1	4
INC X	$X \leftarrow X + 1$	0x75	1	4
INC [expr]	$SRAM[expr] \leftarrow SRAM[expr] + 1$	0x76	2	7
INC [X+expr]	$SRAM[X+expr] \leftarrow SRAM[X+expr] + 1$	0x77	2	8

Table 2.11: **DEC** instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
DEC A	$A \leftarrow A - 1$	0x78	1	4
DEC X	$X \leftarrow X - 1$	0x79	1	4
DEC [expr]	$SRAM[expr] \leftarrow SRAM[expr] - 1$	0x7A	2	7
DEC [X+expr]	$SRAM[X+expr] \leftarrow SRAM[X+expr] - 1$	0x7B	2	8

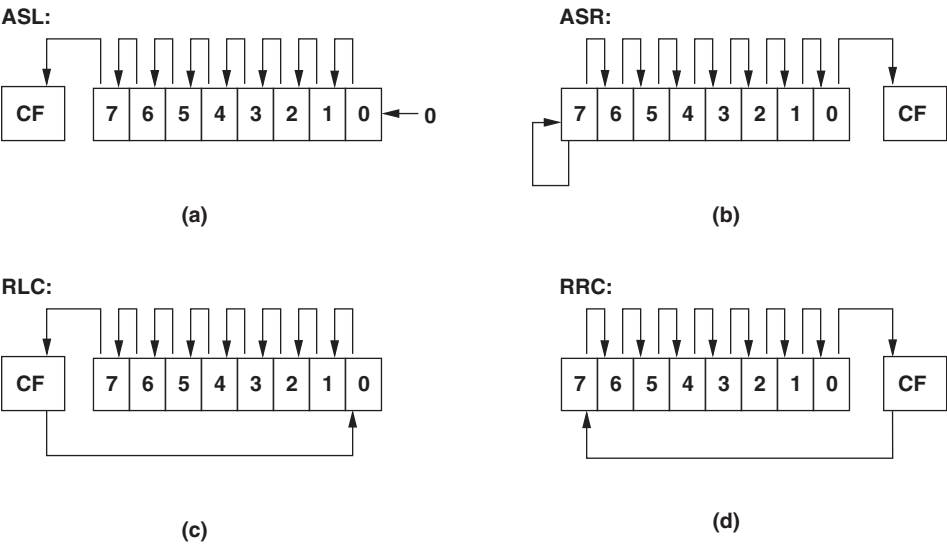


Figure 2.10: Shift and rotate instruction.

Table 2.12: **CMP** instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
CMP A, expr	A - expr	0x39	2	5
CMP A, [expr]	A - SRAM[expr]	0x3A	2	7
CMP A, [X+expr]	A - SRAM[X+expr]	0x3B	2	8
CMP [expr₁], expr₂	SRAM[expr₁] - expr₂	0x3C	3	8
CMP [X+expr₁], expr₂	SRAM[X+expr₁] - expr₂	0x3D	3	9

B.4 Instruction for Comparison: CMP Instructions

The goal of *CMP* instructions is to set the CF and ZF flags. This instruction subtracts the second operand of the instruction from the first operand, but the value is not stored. If the result of the subtraction is negative, the CF flag is set to one, otherwise the flag is cleared. If the result of the subtraction is zero the ZF flag is set to one. If not, the ZF flag is reset to zero. Table 2.12 is a summary of the main attributes of the *CMP* instruction.

B.5 Instruction for Shifting and Rotation: ASL, ASR, RLC, and RRC Instructions

The shift and rotate instructions modify the values in the A register and SRAM. Tables 2.13 and 2.14 list the six kinds of arithmetic shift and rotate instructions, and shows the semantics of the instructions, their opcodes, the number of memory bytes, and their respective execution times.

Figure 2.10 illustrates the semantics of the shift and rotate instructions:

- Arithmetic shift left, ASL, moves the bits one position to the left, as shown in Figure 2.10(a). Bit 7 is pushed into the Carry Flag, and bit 0 is loaded with the bit value 0.
- Arithmetic shift right, ASR, moves the bits one position to the right, as indicated in Figure 2.10(b). Bit 0 is pushed into the Carry Flag, and bit 7 remains unchanged.
- Rotate left through Carry, RLC, moves the bits one position to the left, as shown in Figure 2.10(c). Bit 7 is pushed into the Carry Flag, and bit 0 is loaded with the Carry Flag.
- Rotate right through Carry, (RRC), moves the bits one position to the right, as illustrated in Figure 2.10(d). Bit 7 is loaded with the Carry Flag, and bit 0 is pushed into the Carry Flag.

The Carry Flag is set as shown in Figure 2.10. The Zero Flag is set, if the results of the arithmetic shift and rotate instructions are zero.

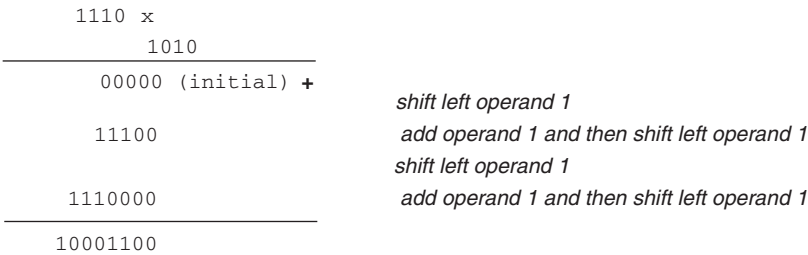
Table 2.13: **ASL** and **ASR** instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
ASL A	see in Figure 2.10	0x64	1	4
ASL [expr]	see in Figure 2.10	0x65	2	7
ASL [X+expr]	see in Figure 2.10	0x66	2	8
ASR A	see in Figure 2.10	0x67	1	4
ASR [expr]	see in Figure 2.10	0x68	2	7
ASR [X+expr]	see in Figure 2.10	0x69	2	8

Example (Multiplication of two four-bit unsigned values). Figure 2.11(a) illustrates the algorithm for multiplying the decimal value 14 (represented by the 4-bit long, binary bitstring 1110) with the decimal value 10 (represented by the 4-bit long, binary bitstring 1010). The result is 8-bits long and has the value 10001100 (i.e., the decimal value 140). In general, if two M-bit unsigned numbers are multiplied, the result is 2 M bits long.

This algorithm uses an auxiliary variable to store the final product and examines the second operand bitwise, starting from the least significant bit. If the analyzed bit is 0 then the first operand is shifted to the left by one position, which is equivalent to multiplying it by 2. If the analyzed bit is ‘1’ then the value of the first operand is added to the auxiliary variable holding the partial result, and the first operand is shifted by one position. After all bits of the second operand have been examined, the auxiliary variable holds the result of the product.

Figure 2.11(b) shows the assembly language source code for the corresponding multiplication routine. The two operands for the multiplication routine `_mul8` are passed to the A and X registers. Because the contents of these registers change during execution, the routine begins by saving the two operands on the stack. Then, the X register is set to point to the top of the stack. The A register, initialized to zero, stores the partial results of the multiplication algorithm. The `CMP` instruction, at address 03CC, is used to determine if the multiplication algorithm has finished, which is indicated by the value of the second operand being zero. Note that the second operand always becomes zero after performing a number of rotate right shifts equal to the position of its most significant nonzero bit. In the case, where the algorithm did not finish, it resets the carry bit, that is bit 2 of the `CPU_F` register, by performing a bitwise AND of the flags register, `CPU_F`, with the value 251 representing the bitvector 11111011 (address 03D1). Then, the second operand is shifted right with a carry. This pushes the next bit to be examined into the carry bit of the flags register, `CPU_F`. If there is no carry, then the currently analyzed bit of the second operand is 0, and the first operand is shifted to the left by one position (address 03D9). This instruction is equivalent to multiplying the second operand by 2. If there is a carry (i.e., the currently analyzed bit) if the second operand is 1, then the content of the first operand is added to the A register holding the partial results. The algorithm iterates until the stop condition at address 03CC is met.



(a)

```
__mul8:
03C7:  PUSH  X
03C8:  PUSH  A
03C9:  MOV   X, SP
03CA:  MOV   A, 0
03CC:  CMP   [X-1], 0
03CF:  JZ    0x03DD
03D1:  AND   F, 251
03D3:  RRC   [X-1]
03D5:  JNC   0x03D9
03D7:  ADD   A, [X-2]
03D9:  ASL   [X-2]
03DB:  JMP   0x03CC
03DD:  ADD   SP, 254
03DF:  RET
```

(b)

Figure 2.11: Assembly code for multiplication.

Table 2.14: **RLC** and **RRC** instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
RLC A	See Figure 2.10	0x6A	1	4
RLC [expr]	See Figure 2.10	0x6B	2	7
RLC [X+expr]	See Figure 2.10	0x6C	2	8
RRC A	See Figure 2.10	0x6D	1	4
RRC [expr]	See Figure 2.10	0x6E	2	7
RRC [X+expr]	See Figure 2.10	0x6F	2	8

C. Instructions for Logical Operations

Logical operations are performed by instructions, for example a bitwise *AND*, bitwise *OR*, bitwise exclusive-*OR* (*XOR*), and bitwise complement (*CPL*). The *TST* instruction performs a bitwise *AND* operation, but its purpose is only to set the *ZF* flag, in as much as the result is not stored. Logical operations can be performed on the flags register, *CPU_F*, to set the *CF* and *ZF* flags to the desired values.

The *AND* instructions perform a bitwise *AND* on the corresponding bits of their two operands, for example a bitwise *AND* of bits zero of the two operands, bitwise *AND* of bits one of the operands, and so on. Table 2.15 lists the different kinds of bitwise *AND* instructions and is a summary of the semantics of the instructions, their opcode, the number of bytes used to represent the instruction in the memory, and the number of clock cycles required for execution. Rows 1-7 in the table show *AND* instructions utilizing different methods of addressing their source and destination, such as source-immediate, source-direct, source-indexed addressing, destination-direct and destination indexed. Row 8 shows the instruction *AND REG[expr₁], expr₂* which has, as an operand, the value in register address *expr₁*. Similarly, the instruction in row 9 uses an operand stored at address “X + *expr₁*” in the register space. Finally, row 10 shows a bitwise *AND* instruction applied to the flags register (register *CPU_F*) and the bit value represented by *expr*. This instruction is used to set, or clear, the *M8C* flags.

The bitwise *OR* instructions compute the bitwise logical *OR* of the corresponding bits of the two operands. For example, bit 0 of the result is the logical *OR* of the bits 0 (least significant bits) of the two operands. Bit 1 of the result is the logical *OR* of the bit 1 of the two operands, etc. Table 2.16 shows the different kinds of *OR* instructions, their semantics, opcode, number of bytes occupied in memory, and number of clock cycles required. Rows 1-7 in the table show *OR* instructions using a different source and destination addressing modes. Rows 8 and 9 contain *OR* instructions with one of the operands located in the register space. The two instructions utilize direct and indexed addressing, respectively. Row 10 presents bitwise *OR* instruction involving the flags register, *CPU_F*. This instruction is the only bitwise *OR* instruction that can modify the *CF* flag, that is bit 2 in register *CPU_F*. This instruction can also change the *ZF* flag which is bit 1 of the *CPU_F* register. In addition, the *ZF* is set whenever a bitwise *OR* instruction produces a result with all its bits being zero. Otherwise, the *ZF* flag is cleared.

The *XOR* instructions compute, bitwise, the exclusive-*OR* (*XOR*) *X* of the corresponding bits of the *XOR* instruction. For example, bit 0 of the result is the logical *XOR* of bit 0 of operand one and bit 0 of the second operand. Table 2.17 is a summary of the different types of *XOR* instructions. As in the case of *AND* and *OR* instructions, *XOR* instructions employ source-immediate, direct, and indexed addressing, and destination-direct and indexed addressing, as shown by the instructions in rows 1-7 of the table. Rows 8 and 9 show *XOR* instructions operating on operands stored in the register space. Finally, row 10 presents the *XOR* instruction with the flags register, *CPU_F*, as one of the operands. *XOR F* affects the *CF* flag, but the instructions in rows 1-9 do not change the flag. The *ZF* flag is set if the result is zero, otherwise the flag is cleared. In addition, *XOR F, expr* can also modify the *ZF* flag.

The *CPL* instruction stores the bitwise complement of the value initially found in the *A* register, in the *A* register. Table 2.18 is a summary of the characteristics of this instruction. Although

Table 2.15: Bitwise AND instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
AND A, expr	$A \leftarrow A \ \& \ \text{expr}$	0x21	2	4
AND A, [expr]	$A \leftarrow A \ \& \ \text{SRAM}[\text{expr}]$	0x22	2	6
AND A, [X+expr]	$A \leftarrow A \ \& \ \text{SRAM}[X+\text{expr}]$	0x23	2	7
AND [expr], A	$\text{SRAM}[\text{expr}] \leftarrow \text{SRAM}[\text{expr}] \ \& \ A$	0x24	2	7
AND [X+expr], A	$\text{SRAM}[X+\text{expr}] \leftarrow \text{SRAM}[X+\text{expr}] \ \& \ A$	0x25	2	8
AND [expr ₁], expr ₂	$\text{SRAM}[\text{expr}_1] \leftarrow \text{SRAM}[\text{expr}_1] \ \& \ \text{expr}_2$	0x26	3	9
AND [X+expr ₁], expr ₂	$\text{SRAM}[X+\text{expr}_1] \leftarrow \text{SRAM}[X+\text{expr}_1] \ \& \ \text{expr}_2$	0x27	3	10
AND REG[expr ₁], expr ₂	$\text{REG}[\text{expr}_1] \leftarrow \text{REG}[\text{expr}_1] \ \& \ \text{expr}_2$	0x41	3	9
AND REG[X+expr ₁], expr ₂	$\text{REG}[X+\text{expr}_1] \leftarrow \text{REG}[X+\text{expr}_1] \ \& \ \text{expr}_2$	0x42	3	10
AND F, expr	$F \leftarrow F \ \& \ \text{expr}$	0x70	2	4

Table 2.16: Bitwise **OR** instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
OR A, expr	$A \leftarrow A \text{ --- } \text{expr}$	0x29	2	4
OR A, [expr]	$A \leftarrow A \text{ --- } \text{SRAM}[\text{expr}]$	0x2A	2	6
OR A, [X+expr]	$A \leftarrow A \text{ --- } \text{SRAM}[X + \text{expr}]$	0x2B	2	7
OR [expr], A	$\text{SRAM}[\text{expr}] \leftarrow \text{SRAM}[\text{expr}] \text{ --- } A$	0x2C	2	7
OR [X+expr], A	$\text{SRAM}[X + \text{expr}] \leftarrow \text{SRAM}[X + \text{expr}] \text{ --- } A$	0x2D	2	8
OR [expr₁], expr₂	$\text{SRAM}[\text{expr}_1] \leftarrow \text{SRAM}[\text{expr}_1] \text{ --- } \text{expr}_2$	0x2E	3	9
OR [X+expr₁], expr₂	$\text{SRAM}[X + \text{expr}_1] \leftarrow \text{SRAM}[X + \text{expr}_1] \text{ --- } \text{expr}_2$	0x2F	3	10
OR REG[expr₁], expr₂	$\text{REG}[\text{expr}_1] \leftarrow \text{REG}[\text{expr}_1] \text{ --- } \text{expr}_2$	0x43	3	9
OR REG[X+expr₁], expr₂	$\text{REG}[X + \text{expr}_1] \leftarrow \text{REG}[X + \text{expr}_1] \text{ --- } \text{expr}_2$	0x44	3	10
OR F, expr	$\text{CPU_F} \leftarrow \text{CPU_F} \text{ --- } \text{expr}$	0x71	2	4

Table 2.17: Bitwise XOR instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
XOR A, expr	$A \leftarrow A \otimes \text{expr}$	0x31	2	4
XOR A, [expr]	$A \leftarrow A \otimes \text{SRAM}[\text{expr}]$	0x32	2	6
XOR A, [X+expr]	$A \leftarrow A \otimes \text{SRAM}[X+\text{expr}]$	0x33	2	7
XOR [expr], A	$\text{SRAM}[\text{expr}] \leftarrow \text{SRAM}[\text{expr}] \otimes A$	0x34	2	7
XOR [X+expr], A	$\text{SRAM}[X+\text{expr}] \leftarrow \text{SRAM}[X+\text{expr}] \otimes A$	0x35	2	8
XOR [expr ₁], expr ₂	$\text{SRAM}[\text{expr}_1] \leftarrow \text{SRAM}[\text{expr}_1] \otimes \text{expr}_2$	0x36	3	9
XOR [X+expr ₁], expr ₂	$\text{SRAM}[X+\text{expr}_1] \leftarrow \text{SRAM}[X+\text{expr}_1] \otimes \text{expr}_2$	0x37	3	10
XOR REG[expr ₁], expr ₂	$\text{REG}[\text{expr}_1] \leftarrow \text{REG}[\text{expr}_1] \otimes \text{expr}_2$	0x45	3	9
XOR REG[X+expr ₁], expr ₂	$\text{REG}[X+\text{expr}_1] \leftarrow \text{REG}[X+\text{expr}_1] \otimes \text{expr}_2$	0x46	3	10
XOR F, expr	$\text{CPU_F} \leftarrow \text{CPU_F} \otimes \text{expr}$	0x72	2	4

Table 2.18: **CPL** instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
CPL A	$A \leftarrow \bar{A}$	0x73	1	4

Table 2.19: **TST** instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
TST [expr₁],expr₂	SRAM[expr₁] & expr₂	0x47	3	8
TST [X+expr₁],expr₂	SRAM[X+expr₁] & expr₂	0x48	3	9
TST REG[expr₁],expr₂	reg[expr₁] & expr₂	0x49	3	9
TST REG[X+expr₁],expr₂	reg[X+expr₁] & expr₂	0x4A	3	10

CF is not modified by this instruction, the ZF flag is set if the resulting value, after complement, is zero.

The *TST* instructions (test with mask) calculate a bitwise *AND* of their operands, but do not store the result. Only the ZF flag is modified as a result of this instruction, but it does not modify the CF flag. Table 2.19 summarizes the *TST* instructions.

Example (Different Bit Manipulations). Figures 2.12(a), (b), and (c) present code that sets, resets, and inverts a single bit in a data byte, respectively. Examples (d), (e), and (f) set, reset, and invert a group of bits, respectively.

Example (a) sets bit 2 of the A register to 1, leaving all other bits unchanged. Mask *MASK_1* has 0 bits for all positions that remain unchanged, and a 1 for the position that is set. A bitwise *OR* instructions sets the desired bit to one.

Example (b) resets bit 2 of the A register leaving all other bits unmodified. The used mask has a bit 1 for all the positions that are not affected, and a bit 0 for the position that is reset. A bitwise *AND* instruction resets the desired bit to zero.

Example (c) complements the third bit of the A register, but other bits are unmodified. The used mask is similar to the mask for example (a). A bitwise *XOR* instruction complements the third bit.

Example (d) sets a group of four bits, bits 2-5, to 1111. The mask has a bit value of 1 for all the positions that are set, and a bit 0 for each unchanged positions. Similar to case (a), a bitwise *OR* instruction is used.

Example (e) resets bits 2 and 4-5, but the other bits are unchanged. The mask used has a bit value of 1 corresponding to the positions that are not modified, and a bit value of 0 for the positions that are reset. The bitwise *AND* instruction resets the wanted bits.

Example (f) complements bits 0-1 and bits 5-6 of the A register. An *XOR* instruction is used with a mask that has a bit value of 1 for all positions that are inverted, and the remaining positions have each a bit value of 0.

<p>MASK_1: ORG 04h ;Mask "00000100"</p> <p> OR A, MASK_1</p> <p style="text-align: center;">(a)</p>	<p>MASK_2: ORG FBh ;Mask "111111011"</p> <p> AND A, MASK_2</p> <p style="text-align: center;">(b)</p>
<p>MASK_3: ORG 04h ;Mask "00000100"</p> <p> XOR A, MASK_3</p> <p style="text-align: center;">(c)</p>	<p>MASK_4: ORG 3Ch ;MASK "00111100"</p> <p> OR A, MASK_4</p> <p style="text-align: center;">(d)</p>
<p>MASK_5: ORG CBh ;Mask "11001011"</p> <p> AND A, MASK_5</p> <p style="text-align: center;">(e)</p>	<p>MASK_6: ORG 63h ;Mask "01100011"</p> <p> XOR A, MASK_6</p> <p style="text-align: center;">(f)</p>

Figure 2.12: Bit manipulations.

D. Instructions for Flow Control

The instructions in this group define the execution flow of programs beyond the sequential execution of instructions. This group of instructions includes *JACC*, *JC*, *JMP*, *JNC*, *JNZ*, *JZ*, *LJMP*, *CALL*, *LCALL*, *RET*, *RETI*, and *SSC* instructions. Table 2.20 is a summary of the M8C's *JMP* instructions.

- *JACC expr*: This is an unconditional jump instruction to the address represented by (PC + 1) plus the contents of the A register plus the value of *expr* expressed as a 12-bit, two's complement value. The accumulator is unaffected by this instruction.
- *JC expr*: If the Carry Flag (CF) is set, it causes program execution to jump to the address represented by (PC + 1) plus the value of *expr* expressed as a 12-bit, two's complement value. The current value of the PC register points to the first byte of the *JC* instruction.

Table 2.20: Jump instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
JACC expr	$PC \leftarrow PC + 1 + A + \text{expr}$	0xEx	2	7
JC expr	$PC \leftarrow PC + 1 + \text{expr}$ $-2048 \leq \text{expr} \leq 2047$	0xCx	2	5
JMP expr	$PC \leftarrow PC + 1 + \text{expr}$ $-2048 \leq \text{expr} \leq 2047$	0x8x	2	5
JNC expr	$PC \leftarrow PC + 1 + \text{expr}$ $-2048 \leq \text{expr} \leq 2047$	0xDx	2	5
JNZ expr	$PC \leftarrow PC + 1 + \text{expr}$ $-2048 \leq \text{expr} \leq 2047$	0xBx	2	5
JZ expr	$PC \leftarrow PC + 1 + \text{expr}$ $-2048 \leq \text{expr} \leq 2047$	0xAx	2	5
LJMP expr	$PC \leftarrow \text{expr}$ $0 \leq \text{expr} \leq 65535$	0x7D	3	7

- *JMP expr*: This instruction causes an unconditional jump to the address represented by $(PC + 1)$ plus the value of *expr* expressed as a 12-bit, two's complement value. The current value of the PC register points to the first byte of the *JC* instruction.
- *JNC expr*: If the CF flag is not set, program execution jumps to the address represented by $(PC + 1)$ plus the value of *expr* expressed as a 12-bit, two's complement value. The current value of the PC register points to the first byte of the *JNC* instruction.
- *JNZ expr*: If the zero flag (ZF flag) is not set, the instruction causes program execution to jump to the address represented by $(PC + 1)$ plus the value of *expr* expressed as a 12-bit, two's complement value. The current value of the PC register points to the first byte of the *JNZ* instruction.
- *JZ expr*: If the ZF flag is set, program execution jumps to the address represented by $(PC + 1)$ plus the value of *expr* expressed as a 12-bit, two's complement value. The current value of the PC register points to the first byte of the *JZ* instruction.
- *LJMP expr*: This instruction causes an unconditional long jump, that is to the 16 bit address represented by *expr*.

The *CALL* instruction causes the PC register to be loaded with the sum of $PC+2$ and *expr*. The current value of PC points to the first byte of the *CALL* instruction and *expr* is a 12-bit, signed number expressed in two's complement form. The current value of the Program Counter is pushed onto the stack, by first pushing the more significant byte followed by the less significant byte. CF and ZF are not changed by this instruction. Table 2.21 shows the main characteristics of the *CALL* instruction.

The *LCALL* instruction loads the value of *expr* into the PC register expressed as a 16-bit, unsigned value of the physical address. The address of the instruction following the *LCALL* instruction is pushed onto the stack. This instruction is at an address pointed to by the current $PC + 2$. The more significant byte of the address is saved on the stack, followed by the less

Table 2.21: **CALL** and **LCALL** instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
CALL <i>expr</i>	$PC \leftarrow PC + 2 + \text{expr}$ $-2048 \leq \text{expr} \leq 2047$	0x9x	2	11
LCALL <i>expr</i>	$PC \leftarrow \text{expr}$ $0 \leq \text{expr} \leq 65535$	0x7C	3	13

significant byte. The CF and ZF flags are not affected by the execution of this instruction. The *LCALL* instruction is summarized in Table 2.21.

Table 2.22: **RET** and **RETI** instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
RET	$SP \leftarrow SP - 1; PC[7:0] \leftarrow SRAM[SP];$ $SP \leftarrow SP - 1; PC[15:8] \leftarrow SRAM[SP];$	0x7F	1	8
RETI	$SP \leftarrow SP - 1; CPU_F \leftarrow SRAM[SP];$ $SP \leftarrow SP - 1; PC[7:0] \leftarrow SRAM[SP];$ $SP \leftarrow SP - 1; PC[15:8] \leftarrow SRAM[SP];$	0x7E	1	10

RET instructions are used to return from a routine called by the instructions *CALL* and *LCALL*. The Program Counter is restored from the stack, and program execution resumes at the next instruction following the call instruction. This instruction does not modify the CF and ZF flags. Table 2.22 is a summary of the *RET* instructions.

RETI instructions are used to return from interrupt service routines (ISRs) and system supervisory calls (SSCs). After the *CPU_F* register is restored from the stack, the CF and ZF flags are updated to the new value of the flags register. Then, the Program Counter is restored. Table 2.22 is a summary of the characteristics of *RETI* instructions.

Example (Calling assembly code routines from C programs). This example discusses the assembly code that corresponds to a function call in a high-level programming language, such as C language. This example is important in that it illustrates the structure required for assembly code routines that are called from programs in high-level languages.

Figure 2.13(a) shows an arbitrary function *f*, with three parameters *a*, *b*, and *c*. Each of these parameters is of type char and *f* returns a value of type char. The type char was selected for simplicity, so that each parameter can be stored as one byte. In addition, the function has two local variables *loc1* and *loc2*. Function *f* is called using the parameters *v1*, *v2*, and *v3*, and the returned value is assigned to variable *d*. The example illustrates the use of *PUSH* and *POP* instructions to transfer data between the calling environment and the called function. The

transferred data includes the parameters of the function, the returned value of the function, and the return address.

Figure 2.13(c) presents the stack structure, called the *activation record*, AR that is created when function f is called. The bottom of AR is pointed to by the X register, so that its different fields can be easily accessed by source-indexed addressing. The bottom entry contains the return value of the function. The old value of the X register is stored on top of it, so that it can be retrieved after returning from the function call. Next, AR contains entries for the three parameters of the function. Then, the return address (the address of the instruction following the function call) is saved on the stack, with the less significant byte sitting on top of the more significant byte. Finally, the two top entries store the values of the local variables *loc1* and *loc2*.

Figure 2.13(b) shows the assembly language code for the function call and the called function f . The calling environment first saves the value of the X register. Then, the X register is loaded with the value of the stack pointer SP, resulting in the X register pointing to the bottom of AR. Instruction 3 increases the stack pointer value thereby allocating a stack entry for the returned value of function f . Because the returned values are of type char, one stack entry (one byte) is sufficient. Instructions 4-9 push the values of the three parameters on the stack. Instruction 10 calls function f and as a result, the address of the instruction following the call is saved on the stack.

The assembly language source code for function f starts at instruction 11. The first two *PUSH* instructions reserve stack entries for the two local variables. Instruction 13 stores a value to the local variable *loc2*, and instruction 14 to variable *loc1*. Instruction 15 assigns a value to the returned value of function f . Before returning from the function call, instructions 16-17 remove the two entries for the local variables from the stack. Instruction RET finds the return address on top of the stack, and the PC register is loaded with the address of instruction 19.

After returning from the function call, the POP instructions, in lines 19–21, remove the entries for the three function parameters from the stack. Instruction 22 pops the returned value of the function, and stores the value in the memory location for variable d . The last instruction restores the X register to its original value prior to the function call.

Table 2.23: **SSC** instructions [2].

Instruction	Semantics	Opcode	Bytes	Cycles
SSC	$\text{SRAM}[\text{SP}] \leftarrow \text{PC}[\text{15:8}]; \text{SP} \leftarrow \text{SP} + 1;$ $\text{SRAM}[\text{SP}] \leftarrow \text{PC}[\text{7:0}]; \text{SP} \leftarrow \text{SP} + 1;$ $\text{SRAM}[\text{SP}] \leftarrow \text{CPU_F}; \text{PC} \leftarrow 0\text{x0000}; \text{CPU_F} \leftarrow 0\text{x00}$	0x00	1	15

System supervisory calls (SSCs) are used to call predefined routines that are stored in ROM and used to perform system functions. Initially, the more significant byte of the *PC* register is saved on the stack, followed by the less significant *PC* byte. Next, the flags register is pushed

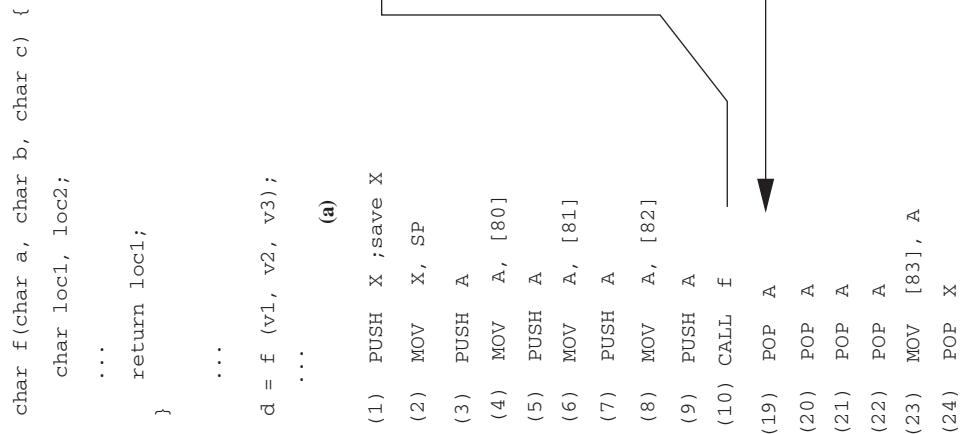


Figure 2.13: Stack-based operations.

onto the stack. After being saved, the flags register is reset. Program execution returns from an *SSC* by executing an *RETI* instruction. Table 2.23 is a summary of the *SSC* instructions.

E. Other Instructions

The *HALT* instruction halts program execution by the microcontroller, and suspends further microcontroller activity pending a hardware reset, for example Power-On, Watchdog Timer, or External Reset. Table 2.24 is a summary of the main characteristics of the *HALT* instruction.

The *NOP* instruction is executed in four clock cycles but has no effect other than to introduce a quantifiable delay in terms of program execution time. It does increment the program counter but does not effect anything else, for example: the CF and ZF flags are unaffected. Table 2.25 summarizes *NOP* instructions.

Example (Design of a sequence detector). Sequence detectors recognize predefined patterns in a stream of bits and are useful in various data communication protocols. The goal of this example is to design a sequence detector that outputs a bit value of 1 whenever an odd number of 0 bits and an odd number of 1 bits are encountered in the input bit stream.

Figure 2.14(a) presents the interface signals of the sequence detector which has two one-bit inputs and two one-bit outputs. Input *In* is the port for the input stream. Input *Ready* is an external signal, which is reset each time a new bit is available at input *In*. After a new input is read by the sequence detector, the *Ack* signal is generated by the detector and is used to remove the *Ready* signal. After the new input has been processed, the corresponding value of the output, *Out*, is produced by the sequence detector. Figure 2.14(b) illustrates the sequencing of the four interface signals.

The behavior of the sequence detector can be described in terms of a finite state machine (FSM) with four states:

- *State A*: This state is entered if an even number of 0s and an even number of 1s occur in the input stream, or upon receiving a reset signal.
- *State B*: This state is entered if an odd number of 0s and an even number of 1s was present in the input stream.
- *State C*: This state is entered if an even number of 0s and an odd number of 1s occur in the input stream.
- *State D*: This state is entered if an odd number of 0s and an odd number of 1s occur in the input stream.

Figure 2.14(c), the state transition diagram, shows all transitions to new states for each possible state of the detector and each possible input value. If the system is in state *A* and the input bit is 0, then the new state is *B*. If the input bit is 1 the new state is *C*. The state diagram also shows the output bits generated for each transition. Output 0 is produced for the transition from state *A* to state *B*. Output 1 is generated for the transition from state *C* to state *D*. The graphical description in Figure 2.14(c) is characterized by the tables provided in Figure 2.14(d).

The design of the sequence detector is given in Figure 2.15 with the data structures organized as illustrated in Figure 2.15(a). The current state of the detector is stored in the variable

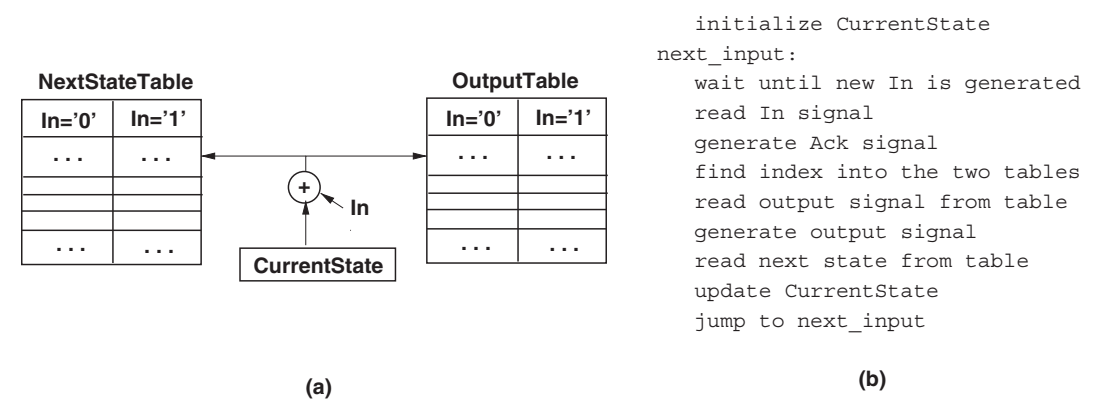


Table 2.26: Performance of the sequence detector implementation.

FSM Step	ROM (bytes)	Execution time (clock cycles)
wait until new input In is generated	6	$15 \times \#_{BW}$
read In signal	6	15
generate Ack signal	6	18
find index into the two tables	7	19
read output signal from table	2	13
generate output signal	4	11
read next state from table	4	18
update CurrentState	2	5
wait for Ready Signal to go High	6	$15 + \#_{BW}$
jump to next input	2	5
Total	45	149 (without busy waiting)

signal value is 1 and then 0. Next, instructions 10–13 compute the index used to access the two tables. Instruction 10 selects the input bit using mask *IN_MSK* (00000010). The A register is shifted one position to the right, so that it can be used for the index. The index value is stored in the SRAM location *TEMP2*. Instruction 14 accesses Table *OutputTable* to find the bit value that has to be output by the detector. Instructions 15–16 set bit 5 of the port register to the required value. Instructions 17–18 access Table *NextStateTable* to get the encoding of the next state. The encoding is stored in the variable *CurrentState* in instruction 19. Instructions 20–22 delay execution until the *READY* signal is one (inactive). Finally, program execution jumps to label FSM to start the processing of the next input bit.

Performance analysis. Table 2.26 is a summary of the memory and timing performance of the implementation. Rows correspond to the steps shown in Figure 2.15(b). The first row corresponds to “busy-waiting” until a new input bit is found at the sequence detector. The timing of the step depends on the number of executed iterations ($\#_{BW}$). The last row shows the amount of memory and the number of clock cycles required to execute the code. Assuming that the $\#_{BW} = 1$, new input data are available each time a new iteration starts. Because one iteration of the code is executed in 149 clock cycles, thus for a clock frequency of 24 MHz, this results in an execution time of $149 \times 0.041 \mu\text{sec}$, or $6.109 \mu\text{sec}$ per iteration. Thus, input bit streams of

$$\frac{1}{6.109 \times 10^{-6}} \text{bits/sec} \approx 163 \text{ kbits/sec} \quad (2.2)$$

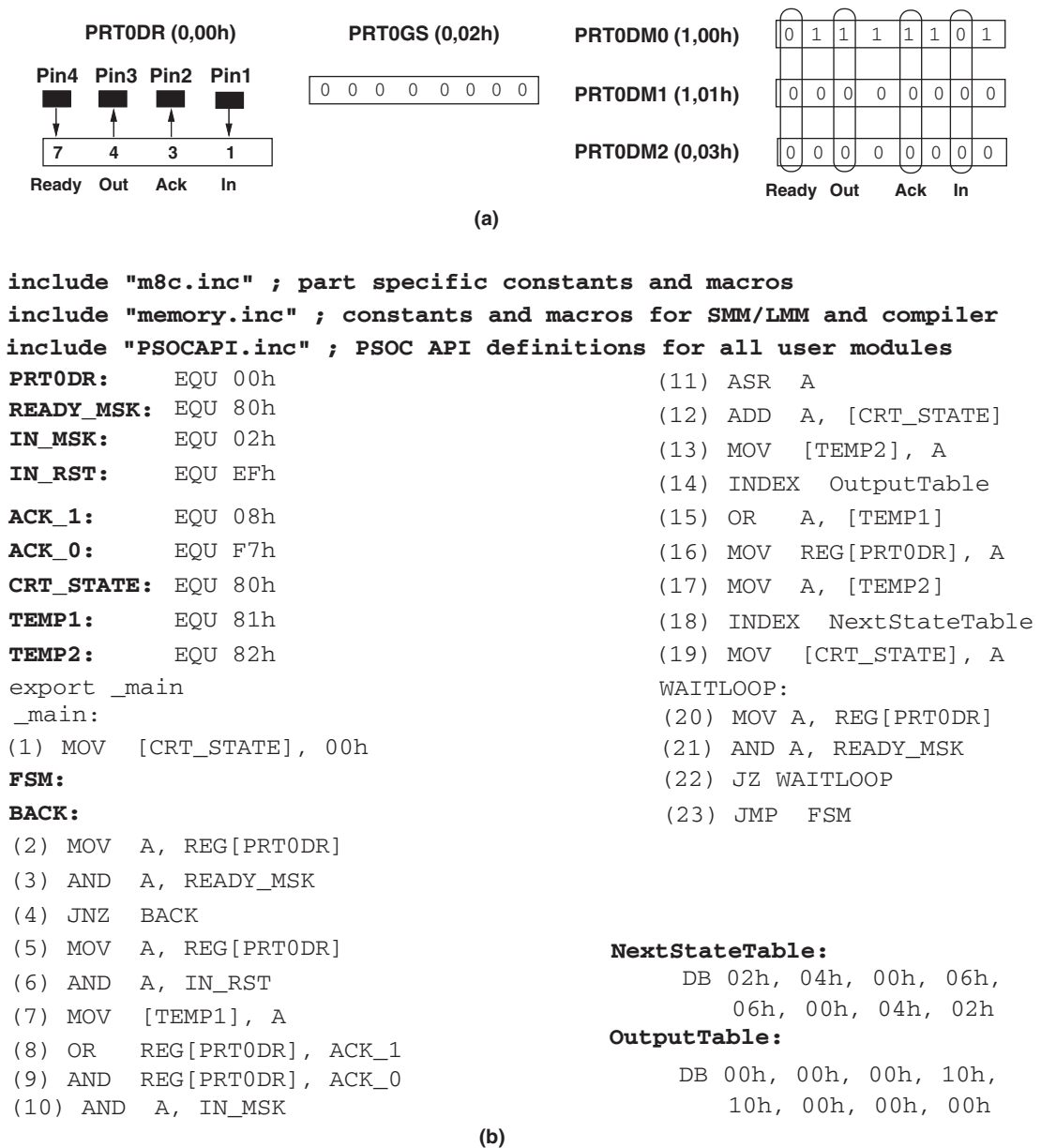


Figure 2.16: Sequence detector implementation using INDEX instructions.

can be processed without losing a bit. For higher input frequencies, some of the data might be lost, because the sequence detector then becomes too slow to process all the inputs.

This analysis also shows that about 33% of the total execution time, excluding busy waiting time, is consumed accessing the two tables, 10% of the time is required to read data from the input port, 12% of the time is required to generate the *Ack* signal, 7% of the time is required to produce the *Out* signal, 3% to update the state variable, and 3% of the time is required to jump to the beginning of the detection code. Thus, speedingup execution for processing faster inputs

```

include "m8c.inc" ; part specific constants and macros
include "memory.inc" ; constants and macros for SMM/LMM and compiler
include "PSOCAPI.inc" ; PSOC API definitions for all user modules
PRT0DR: EQU 00h
READY_MSK: EQU 80h
IN_MSK: EQU 02h
IN_RST: EQU EFh
ACK_1: EQU 08h
ACK_0: EQU F7h
TEMP1: EQU 81h
NextStateTable: EQU 10h
OutputTable: EQU 50h
export _main
_main:
(1) MOV [00h], 00h
; NextStateTable
(2) MOV [10h], 02h
(3) MOV [11h], 04h
(4) MOV [12h], 00h
(5) MOV [13h], 06h
(6) MOV [14h], 06h
(7) MOV [15h], 00h
(8) MOV [16h], 04h
(9) MOV [17h], 02h
; OutputTable
(10) MOV [50h], 00h
(11) MOV [51h], 00h
(12) MOV [52h], 00h
(13) MOV [53h], 10h
(14) MOV [54h], 10h
(15) MOV [55h], 00h
(16) MOV [56h], 00h
(17) MOV [57h], 00h

(18) MOV X, 00h
FSM:
BACK:
(19) MOV A, REG[PRT0DR]
(20) AND A, READY_MSK
(21) JNZ BACK
(22) MOV A, REG[PRT0DR]
(23) AND A, INT_RST
(24) MOV [TEMP1], A
(25) OR REG[PRT0DR], ACK_1
(26) AND REG[PRT0DR], ACK_0
(27) AND A, IN_MSK
(28) ASR A
(29) ADD A, [X+0]
(30) MOV X, A
(31) MOV A, [X+OutputTable]
(32) OR A, [TEMP1]
(33) MOV REG[PRT0DR], A
(34) MOV A, [X+NextStateTable]
(35) MOV [X+0], A
WAITLOOP:
(36) MOV A, REG[PRT0DR]
(37) MOV A, READY_MSK
(38) JZ WAITLOOP
(39) JMP FSM

```

Figure 2.17: RAM-based sequence detector implementation.

can be achieved by two orthogonal improvements: (1) finding faster ways of accessing data in the two tables, and (2) using faster I/O interfaces.

RAM-based implementation. Figure 2.17 presents an alternative implementation, in which the FSM data are first moved from the flash memory to the RAM. Data accessing is simpler and faster than for the example in Figure 2.16(b). Instead of using *INDEX* instructions, the program uses indexed addressing to access the FSM tables. The modifications to the code are shown in bold.

2.2 Memory Space

This section discusses PSoC's SRAM and ROM space.

A. SRAM Space

The PSoC architecture can have as many as eight, 256 -byte, memory pages. The 256 bytes of any given page are addressed by an eight bit address. Different memory pages can be used for storing the data variables of a program, the stack, and the variables accessed by the two indexed addressing modes. Having multiple memory pages also simplifies the process of simultaneous execution of multiple tasks in multitasking applications. The variables of a task can be stored in a separate page, and reducing the amount of data transfer required when switching from the execution of one task to another. On the negative side, the assembly code programming becomes slightly more difficult, because programmers must then manage data stored over multiple SRAM pages.

Figure 2.18 presents the paging mechanism of the SRAM space. Bit 7 of the *CPU_F* register is used to enable and disable the SRAM paging. If the bit is zero then all memory accesses are to page zero. If the bit is set, then the last three bits of the *CUR_PP* register at address 0, D0H (*CUR_PP*[2:0]) select the current SRAM page. Then, with the exception of *PUSH*, *POP*, *CALL*, *LCALL*, *RETI*, *RET*, and *MVI* instructions, all other SRAM accesses refer to this page.

In addition to the current SRAM page, the PSoC architecture also gives the option to setup dedicated pages for the stack and for the indexed addressing mode. The stack page is selected

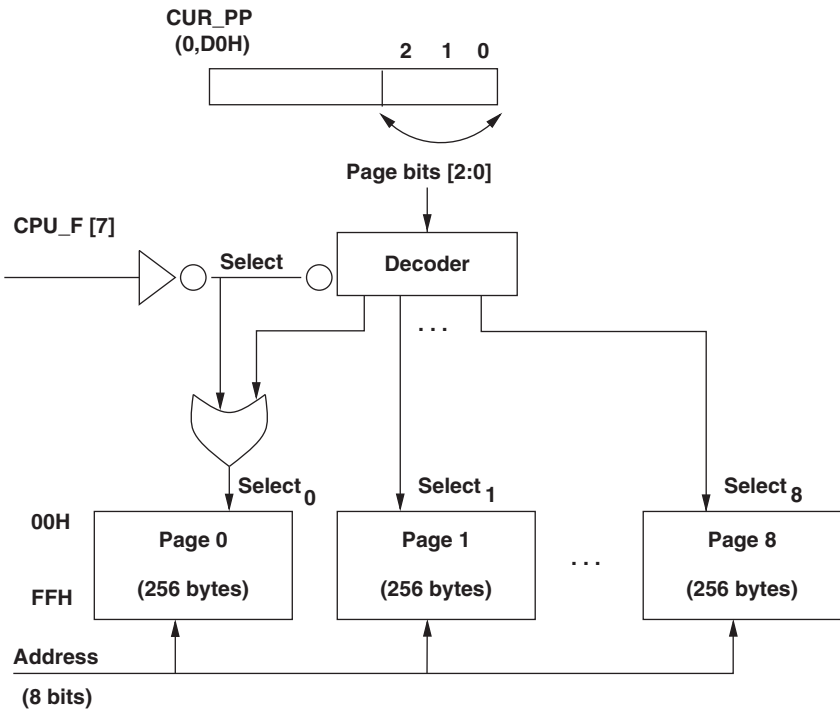


Figure 2.18: Paged SRAM space [4].

by the bits 2–0 of the *STK_PP* register at address 0, D1H. After a reset operation, the three bits are set to 000, and hence all stack operations refer to page 0.

The indexed addressing mode uses one of three possible SRAM pages depending on the value of the two bits 7–6 of the *CPU_F* register at address x, F7H:

- If the bits are 0 then page 0 is used for the indexed addressing mode.
- If the bits are 01 or 11 then the used page is that pointed by the bits 2–0 of the *STK_PP* register at address 0, D1H. Among other situations, this mode is useful to access the elements of a routine’s activation record in the memory stack.
- If the bits are 10 then indexed addressing uses the page pointed by the bits 2–0 of the *IDX_PP* register at address 0, D3H. This mode can be used for transferring data located in different memory pages, for example when data need to be sent from one task to another, assuming that each task has its own memory page.

Four temporary data registers, *TMP_DRx*, are available to help in accessing the data transfer between pages. The registers are always accessible independent of the current SRAM page. The *TMP_DR0* register is at address x, 60H, the *TMP_DR1* register at address x, 61H, the *TMP_DR2* register at address x, 62H, and the *TMP_DR3* register is located at address x, 63H.

B. SROM Space

SROM stores the code of eight routines used in booting the PSoC chip, reading from and writing blocks (64 bytes) to the flash memory, and circuit calibration.

The routines are called by executing an *SSC* instruction that uses the A register for distinguishing among the eight routines (hence, the identity of the called routine is passed as a parameter to the SSC instruction by the A register). In addition, the SROM routines use a set of extra parameters, which are located at predefined memory locations. In addition to the returned values that are returned as part of their functionality, each SROM function also returns a code value that signals the success or failure of the routine execution. Table 2.29 enumerated the four possible return values.

Each flash block is protected against illegal SROM function calls by setting its accessing mode to one of four possible modes: in the unprotected mode, external read and write operations are allowed, as well as internal writes and read operations of entire blocks. Internal read operations are allowed in each of the four modes. The second mode (factory upgrade) allows only external write and internal write operations. The third mode (field upgrade) permits only internal write operations. Finally, the fourth mode (full protection) permits only internal read operations.

Table 2.27 lists the SROM eight functions in column one, the value that needs to be loaded into the A register before executing the SSC routine in column two, and the required stack space in column three. Table 2.28 presents the variables that are used for passing extra parameters to the SROM functions, and the addresses of these variables in the SRAM. Note that all variables are located in page 0. Finally, Table 2.29 enumerates the four codes that indicate the status at the end of an SROM function execution. To distinguish between legal and illegal SROM function calls, the value of parameter *KEY1* must be set to 3AH and the value of parameter *KEY2* to the stack pointer, SP, plus 3 before executing the SSC instruction for the call.

The behavior of the SROM functions is as follows:

Table 2.27: SROM functions [1].

Function name	Function code	Stack space
SWBootReset	00H	0
ReadBlock	01H	7
WriteBlock	02H	10
EraseBlock	03H	9
TableRead	00H	3
Checksum	00H	3
Calibrate0	00H	4
Calibrate1	00H	3

- SROM functions called at system reset.
 - *Checksum*: The function computes the checksum over a number of blocks in the flash memory. The number of blocks is defined by the parameter *BLOCKID* (see Table 2.28). The checksum value is 16 bits long, and is returned using the following parameters: *KEY1* which holds the least significant byte, and *KEY2* which holds the more significant byte of the checksum value.
 - *SWBootReset*: This function is automatically executed upon hardware reset. It initializes some of the CPU registers and page 0 of SRAM. It first verifies the checksum of the calibration data. For valid data, it loads the following registers with the value 00H: A register, X register, CPU_F register, SP register, and the Program Counter, PC. Also, it sets the SRAM page 0 locations to predefined values, and starts execution of the user code at address 0000H in the flash memory.

The SRAM page 0 memory locations are initialized, depending on their addresses, to the value 0x00, some predefined hexadecimal values, or to values programmed by the bit IRAMDIS (bit 0) of the control register, *CPU_SCR1*, at address x,FEH. If the bit is set to 0 then the corresponding SRAM cells must be initialized to 0x00H after a watchdog reset, otherwise not, therefore preserving the SRAM value before reset. More details about the specific values that are loaded into each memory location in the SRAM page 0 can be found in [1].

Table 2.28: SROM function variables [1].

Variable name	SRAM
KEY1 / RETURN CODE	0, F8H
KEY2	0, F9H
BLOCKID	0, FAH
POINTER	0, FBH
CLOCK	0, FCH
Reserved	0, FDH
DELAY	0, FEH
Reserved	0, FFH

Table 2.29: SROM return codes [1].

Return code value	Description
00H	successful completion
01H	function is not allowed because of the protection level
02H	software reset without hardware reset
03H	fatal error

- The following are the SROM functions that read and write to the flash memory:

- *ReadBlock*: This function transfers a data block consisting of 64 bytes from the flash memory to the SRAM. Figure 2.19 presents the pseudocode for this function. First, the function verifies if the block pointed by the parameter *BLOCKID* is readable. If the block is not readable then the A register and the parameters *KEY1* and *KEY2* are loaded with values which indicate that reading was not possible due to protection restrictions.

If the block protection allows reading, then 64 bytes are read from the flash memory using ROMX instructions, and stored in the SRAM by MVI instructions. The SRAM area is pointed to by the parameter *POINTER*. The SRAM page is determined by the MVW_PP register at address 0, D5H, for example for any MVI instruction. The successful completion of the operation is indicated by loading parameters *KEY1* and *KEY2* with the value 00H.

The flash bank is selected by the FLS_PR1 register at address 1, FAH, if the architecture has multiple flash banks. Bits 1–0 of the register select one of the four possible flash banks.

- *TableRead*: This function accesses part-specific data stored in the flash memory. For example, these data are needed to erase or to write blocks in the flash memory. The table to be accessed is selected by programming the parameter *BLOCKID*. Table 0 provides the silicon ID of the chip, tables 1 and 2 the calibration data for different power supplies and room temperatures, and table 3 the calibration data for correctly erasing or writing to the flash memory.
- *EraseBlock*: This function erases a block in the flash memory. The block is indicated by the parameter *BLOCKID*. The function first checks the protection of the block, and, if writing is not enabled, then the value 01H, indicating failure due to protection, is loaded into the parameter *KEY1*.

In addition to the parameter *BLOCKID*, two other parameters, *DELAY* and *CLOCK*, must also be set before calling the *EraseBlock* function. The parameters are introduced in Table 2.28. If the CPU clock is in the range 3–12 MHz, then the value of the parameter *DELAY* is set by the following[1],

$$DELAY = \frac{10^{-4} \times CPU_{speed} \text{ (Hz)} - 84}{13} \quad (2.3)$$

For a higher clock speed, the *DELAY* value is computed as [1]:

$$DELAY = \frac{10^2 \times 12 - 84}{13} \quad (2.4)$$

```

SROM function ReadBlock:
  (1) if the block pointed by BLOCKID
      is NOT readable then
        (2) load 00H into the register A;
        (3) load 00H into KEY2;
        (4) load 01H to KEY1;
        (5) exit;
      else
        (6) read 64 bytes from flash using the
            ROMX instruction and store the
            bytes in SRAM using MVI instructions.
            the SRAM area is pointed by POINTER;
            the flash bank is selected by
            register FLS_PR1.
        (7) load 00H to KEY1;
        (8) load 00H to KEY2;
        (9) exit;
end SROM function ReadBlock.

SROM function WriteBlock:
  (1) if the block pointed by BLOCKID
      is NOT writeable then
        (2) load 00H into the register A;
        (3) load 00H into KEY2;
        (4) load 01H to KEY1;
        (5) exit;
      else
        (6) read 64 bytes from SRAM using the
            MVI instruction and store the
            bytes in the flash memory.
            the SRAM area is pointed by POINTER;
            the flash bank is selected by
            register FLS_PR1.
        (7) load 00H to KEY1;
        (8) load 00H to KEY2;
        (9) exit;
end SROM function WriteBlock.

```

Figure 2.19: ReadBlock and WriteBlock SROM functions [1].

The parameter *CLOCK* is defined as:

$$CLOCK = B - \frac{2 \times M \times T}{256} \quad (2.5)$$

The values for B and M are device specific, and are accessed using the *TableRead* SROM function.

Different values are used for temperatures, T below and above, 0°C . For example, for temperatures below 0°C , the value of M is stored in the flash bank 0 at address F8H, and B at address F9H. For temperatures higher than 0°C , the value of M is found at address FBH and B at address FCH.

- *WriteBlock*: This function writes a data block in the SRAM to the flash memory. Figure 2.19 shows the pseudocode for this function. The parameter *POINTER* points to the address of the block in SRAM, and the parameter *BLOCKID* points to the location in flash memory region where the data are to be copied. The flash bank is determined by bits 1-0 of the FLS.PR1 register at address 1, FAH.

This function first determines if the flash memory block pointed to by the parameter *BLOCKID* is writable, or not. If it is writable, then 64 bytes are accessed by MVI instructions, starting at the SRAM address given by the parameter *PARAMETER*.

Parameters *DELAY* and *CLOCK* must also be programmed for correct writing to the flash memory. The value of *DELAY* is set according to Equation (2.3). The parameter *CLOCK* is defined as [1]:

$$CLOCK = \frac{CLOCK_E \times MULT}{64} \quad (2.6)$$

The value $CLOCK_E$ is the clock parameter for the erase operation, and is computed according to the equation (2.5). The value of the parameter *MULT* depends on the temperature, is stored in table 3 of the flash bank 0 (at address FAH for cold temperatures, and at address FDH for hot temperatures), and can be accessed using the *TableRead* SROM function.

- The functions used in calibration are the following:
 - *Calibrate0*: This function transfers the calibration values from the flash memory to the required registers.
 - *Calibrate1*: This function executes the same functionality as the function *Calibrate0*, but in addition also computes the checksum of the calibration data. If the checksum is incorrect, then a hardware reset is generated, and bit IRESS (bit 7) of the register CPU_SCR1 is set. The CPU_SCR1 register is at address x, FEH.

The parameter *POINTER* points to a 30-byte buffer that is used for computing the checksum. Also, as MVI instructions are used for the actual data transfer, the MVR_PP and MVW_PP registers must point to the same SRAM page.

2.3 Conclusions

This chapter has presented the important characteristics of a microcontroller's instruction set and discussed programming techniques, in assembly language, for six popular applications and routines: data block transfer, stack operation, unsigned data multiplication, calling assembly routines from programs in high-level programming languages, bit manipulations, and sequence detectors. It has also discussed the memory space of embedded architectures, including the

microcontroller registers, the SRAM and nonvolatile memory spaces, and the status and control registers.

PSoC's M8C microcontroller instruction set was discussed in some detail as an illustrative example. PSoC instructions can use ten addressing modes that result from combining the four basic addressing modes: immediate addressing, direct addressing mode, indexed addressing, and indirect addressing. These addressing modes allow a tradeoff between execution speed and code size and provide flexibility in mapping data to the microcontroller memory. Using immediate- and direct-addressing modes results in faster, smaller, but also less flexible code.

M8C's instruction set includes five categories of instructions: instructions for (1) data transfer, (2) arithmetic operations, (3) logic operations, (4) execution flow control, and (5) miscellaneous.

- The instructions for data transfer involve M8C's general-purpose A and X registers, the SRAM space (instructions MOV, MVI, and SWAP), the stack (instructions POP and PUSH), and the nonvolatile memory (instructions ROMX and INDEX).
- The instructions for arithmetic operations perform additions (instructions ADD and ADC), subtractions (instructions SBB and SUB), increment and decrement (instructions INC and DEC), comparison (instruction CMP), arithmetic shift (instructions ASL and ASR), and rotate (instructions RLC and RRC).
- The instructions for logic operations execute bit-level logic AND (instructions AND), OR (instructions OR), XOR (instructions XOR), and complement (instructions CPL).
- The instructions for flow control include: jumps (instructions JACC, JC, JMP, JNC, JNZ, JZ, and LJMP), subroutine calls (instructions CALL, LCALL, RET, and RETI), and system supervisory calls (instructions SSC).
- Other instructions include HALT and NOP instructions.

PSoC's memory space consists of the microcontroller registers (A, X, CPU_F, SP, and PC registers), the paged SRAM space, the nonvolatile memory (flash and ROM), and the register space for programming the mixed-signal architecture. The paged memory includes up to eight SRAM pages, inclusive, and is managed by the CUR_PP (the current memory page), STK_PP (stack page), and IDX_PP (indexed addressing) registers. The temporary registers, TMP_DR, are available for speeding-up the data transfer between pages. Eight system routines are stored in ROM that are called by the instructions SSC. The register space is organized as two register banks, which help fast reconfiguration of the architecture.

Assembly code routines for six different applications have also been provided in this chapter, for example the sequence detector design provides the most comprehensive discussion. The goal of sequence detection is to recognize a predefined bitstring in a sequence of input bits. This example presented the pseudocode of the detector specified as a finite state machine (FSM), two implementation alternatives, and the performance analysis of the implementation. The design used a handshaking mechanism to read input and output results at the general-purpose ports of the architecture. The two implementation alternatives stored the next state and output tables either in the flash memory, or in SRAM. Performance analysis estimated the execution time and the memory required. It also estimated the highest data rate of the input that can be processed without data loss. The example also accounted for the various contributions to the total execution time of the implementation.

2.4 Recommended Exercises

1. Develop Assembly routines for adding and subtracting two unsigned operands of lengths two bytes - in the first case, and then four bytes - in the second case. Many C language compilers represent **short unsigned int** variables using 2 bytes, and **unsigned int** variables using 4 bytes.
2. Write a routine in Assembly language for dividing two unsigned bytes. Extend the routine for division of numbers of arbitrary size (an arbitrary number of bytes is used for their representation).
3. Write assembly language routines for multiplication and division of signed operands of lengths 2, 4, and 8 bytes. Optimize the routines for a minimum number of execution clock cycles.
4. For a bitstring of length M , write Assembly code routines that set and reset a bit at position P . Also, develop routines that set and reset a field of length L starting at position P .
5. Write a set of routines that implement the abstract data type a set of positive integers up to 512 (the elements of the set can be the integers 0, 1, ..., 512). Propose routines for initializing a set as the empty set, adding the value v to the set, eliminating the value v from the set, and verifying if the value v is in the set. Use these routines to compute the reunion, intersection, and difference between two sets.
6. Write a routine in the M8C microcontroller Assembly language for sorting in increasing order a vector of unsigned bytes. Optimize the flexibility of your code, so that it can run for different vector sizes and for different memory regions (memory pages) storing the vector.
7. Develop an Assembly code program that searches a value in a vector of sorted values. The algorithm should implement a binary search method: at the first step, the algorithm compares the searched value with the value at the middle position of the vector. If the searched value is less then the algorithm repeats the process for the lower half, and if the value is larger then the algorithm considers the upper half. The algorithm returns the position of the searched element (if the value was found), or -1 , otherwise. The starting address and the length of the vector are inputs to the procedure. Assume initially that the value is one byte long. Then, discuss the changes that must be operated so that data values of size B bytes are handled by the procedure.
8. Write Assembly code routines for multiplying two bidimensional matrices of unsigned integers. The routines should be flexible in handling matrices with different numbers of rows and columns.
9. Develop the Assembly code routines for a “software” lock. One routine must read serially eight bits at a rate of one bit per second. Then, the routine checks if the bitstring is equal to the preset code of the lock. If the two are equal than the signal *Unlock* is set to 1, otherwise it is set to 0. The second routine sets the code of the lock by reading serially eight bits at a rate of ten bits per second.
10. Develop an Assembly language routine that implements a palindrome recognizer. A palindrome is a bitstring which is equal to its reversed form. For example, the bitstring 111000111 is

a palindrom, but the bitstring 10100011 is not. Upon receiving the pulse *START*, the routine reads eight bits that are sent serially at the bit rate T . The value T is programmable. The eight bits indicate the length L of the bitstring that is checked. Then, the L bits of the bitstring are serially input at the same rate T . The system outputs bit 1 if the bitstring is a palindrom, and bit 0, otherwise.

11. Develop Assembly routines that implement a traffic-light controller. The traffic light controller controls the traffic moving along four directions: south-to-north, north-to-south, east-to-west, and west-to-east. The traffic light should stay in a state (e.g., green light) for a specified time T , and then move to the next state. If no cars are passing on a direction (e.g., north-to-south and south-to-north, or east-to-west and west-to-east) for B consecutive states then the value of the time interval T for the state “red light” should be shortened to $T/2$, and then to $T/4$, and so on, until it falls below the threshold T_H , and when it is set to 0. Once cars move again in that direction the value of the state “red light” should be set back to the value T . The moving of cars in a direction is signaled to the controller by proximity sensors, which generate an asynchronous pulse of width T_{pulse} .

12. The task is to develop Assembly code for a bottled-water vending machine. A bottle costs \$0.45. The vending machine accepts nickels, dimes, and quarters. The customer begins entering coins, one at a time. If exact change has been entered then the output signal *Unlatch* is generated, so that the customer can get a bottle. If the amount of deposited money exceeds the price of a bottle, then change is given. If there is not enough change in the repository then the coins are refunded by issuing the output signal *Refund*, and the signal *Unlatch* is not generated. If there is enough change in the repository then it should be paid using the minimum amount of coins. The repository indicates to the controller the number of coins of each type available in the repository: bits Nx show the number of nickels, bits Dx the number of dimes, and bits Qx the number of quarters. Change is given to the customer by generating pulse signals for the repository: the signal NR releases one nickel, the signal DR releases one dime, and the signal QR releases one quarter. One coin is released at a time. (*Acknowledgment:* The exercise is based on a homework problem for Prof. R. Vemuri’s VLSI design course at the University of Cincinnati).

13. Write Assembly routines that implement the basic operators of a circular FIFO (first input first output) buffer. The buffer is described by two pointers. The pointer head indicates the first element of the buffer, and the pointer tail points to the last element of the buffer. A new element is inserted at the tail of the buffer, and only the value that is pointed by the pointer head can be removed from the buffer. The FIFO buffer is of size MAX - where the value of MAX is programmable. Once the pointer *head* reaches the value MAX then it is set back to the first element of the buffer. Similarly, once the pointer *tail* indicates to the first buffer element, the pointer is set to the last element in the FIFO buffer. The assembly routines should offer the following functionality: create a new FIFO buffer starting at address xxH , reset the buffer, introduce an element into the buffer, and remove a value from the buffer.

14. Write an Assembly code routine that calculates the expression $((4 + 5) * 3) * (2 + 3)$ using only the stack for storing the operand values and the intermediate results of the expression.

15. Develop Assembly code routines for implementing the abstract data type, sorted linked lists of unsigned bytes. Define the routines for setting up an empty list, inserting a new element to the list, and removing a value from the list.

16. For the routines in Figure 2.8, identify all situations in which the “safe ” pop and push routines are not safe anymore. Extend the two routines so that they are safe in all situations. Explain your procedure for identifying the conditions that lead to unsafe operation.

17. Write an Assembly routine for realizing the behavior of a pulse width modulation block. The designer should have the option of programming the pulse width α of the PWM routine through calling a C function with two parameters: the new value of the pulse width and a pointer to the PWM block data.

18. Develop the following routines and programs in Assembly code:

- A routine that displays the message “Hello world” on the LCD.
- A program that controls the blinking of an LED once every two seconds with a 50% duty cycle (the LED is on for one second, and off for one second).
- Two assembly routines that convert a binary number to BCD, and BCD to binary number, respectively.
- A routine that implements a frequency calculator.
- A routine for the cyclic redundancy check (CRC) function.

19. For the example shown in Figure 2.13, modify the structure of the activation record and show the Assembly code that must be inserted in the calling and called routines, so that input parameters of any byte size can be passed, and results of any size are returned.

20. For the C compiler that you are using for your embedded applications, analyze the Assembly code that is generated for different C statements, for example assignment, if, case, for, while statements, and different data types, such as arrays and structures. Understand what code optimizations are carried out by the compiler.

21. Develop Assembly code that reads the values of the constants B , M , and $MULT$ specific to your PSoC chip.

22. Using SSC instructions and the routines stored in the ROM memory, implement a program that writes N blocks to the flash memory, then reads M blocks of data from the flash memory to the SRAM memory, and finally, erases the M blocks of flash memory,

23. For the sequence detector design in Figures 2.15 and 2.16, estimate the execution time overhead due to busy waiting for different values of the input bit rates, such as 0.1 kbits/sec, 1 kbits/sec, 10 kbits/sec, 50 kbits/sec, and 100 kbits/sec. Discuss the results and propose a solution to decrease the busy-waiting overhead.

24. For Exercise 14, estimate the execution time in clock cycles if the data values are represented on one, two, and four bytes. Discuss the increase in execution time depending on the size of the values that are processed.

25. Find an analytical formula that predicts the execution time of an algorithm depending on the memory size of the processed values.
26. For exercise 8, assume that each of the matrices occupies multiple SRAM pages. Modify the algorithm in this case, and propose a way of distributing the matrix elements to the SRAM pages, so that the execution time of the program is minimized.

Bibliography

- [1] PSoC Mixed Signal Array, Technical Reference Manual, *Document No. PSoC TRM 1.21*, Cypress Semiconductor Corporation, 2005.
- [2] PSoC Designer: Assembly Language User Guide, *Spec. #38-12004*, Cypress Microsystems, December 8 2003.
- [3] PSoC Designer: C Language Compiler User Guide, *Document #38-12001 Rev.*E*, Cypress Semiconductor, 2005.
- [4] K. Boulos, Large Memory Model Programming for PSoC, *Application Note AN2218*, Cypress Microsystems, September 13 2004.
- [5] J. Byrd, Interfacing Assembly and C Source Files, *Application Note AN2129*, Cypress Microsystems, May 29 2003.
- [6] J. Holmes, Multithreading on the PSoC, *Application Note AN2132*, Cypress Microsystems, February 24 2004.
- [7] D. Lewis, *Fundamentals of Embedded Software. Where C and Assembly Meet*, Saddle River, NJ: Prentice Hall, 2002.
- [8] M. Raaja, Binary to BCD Conversion, *Application Note AN2112*, Cypress Microsystems, February 12 2003.
- [9] W. Snyder, J. Perrin, Flash APIs, *Cypress Microsystems*, August 4 2004.
- [10] S. Sukittanon, S. Dame, Embedded State Machine Design for PSoC using C Programming (Part I of III), *Application Note AN2329*, Cypress Semiconductor, February 8 2006.
- [11] S. Sukittanon, S. Dame, Embedded State Machine Design for PSoC using an Automatic Code Generator (Part II of III), *Application Note AN2332*, Cypress Semiconductor, December 5 2005.
- [12] S. Sukittanon, S. Dame, Embedded State Machine Design for PSoC with Interfacing to a Windows Application (Part III of III), *Application Note AN2333*, Cypress Semiconductor, January 5 2006.
- [13] D. Van Ess, Unsigned Multiplication, *Application Note AN2032*, Cypress Semiconductor, June 21 2002.
- [14] D. Van Ess, Signed Multi-Byte Multiplication, *Application Note AN2038*, Cypress Semiconductor, August 29 2002.
- [15] D. Van Ess, A Circular FIFO, PSoC Style, *Application Note AN2036*, Cypress Microsystems, July 21 2002.

<http://www.springer.com/978-1-4419-7445-7>

Introduction to Mixed-Signal, Embedded Design

Doboli, A.; Currie, E.H.

2011, XXIV, 452 p. 300 illus. With online files/update.,

Hardcover

ISBN: 978-1-4419-7445-7