AVR Microcontroller

Microprocessor Course

Chapter 6

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING

Aban 1401

6.1 : INTRODUCING SOME MORE ASSEMBLER DIRECTIVES

Arithmetic and logic expressions with constant values

The AVR Studio IDE supports logic operations between expressions as well.

Table 6-1: Arithmetic Operators

Symbol	Action
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo

For example, in the following program R21 is loaded with 0x14:

```
.EQU C1 = 0x50
.EQU C2 = 0x10
.EQU C3 = 0x04
LDI R21,(C1&C2)|C3   ;R21=(0x10&0x50)|0x04 = 0x10|0x04= 0x14
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING

6.1 : INTRODUCING SOME MORE ASSEMBLER DIRECTIVES

In Table 6-3 you see the shift operators, which are very useful. They shift left and right a constant value.

Table 6-3: Shift Operators

Symbol	Action	Example
<<	Shifts left the left expression by the number of places given by the right expression	`LDI R20,0b101<<2 ;R20=0b10100`
>>	Shifts right the left expression by the number of places given by the right expression	`LDI R20,0b100>>1 ;R20=0b010`

For example, the following instruction loads theR20 register with 0b00001110:

```
LDI R16,0b00000111<<1 ;R16 = 0b00001110
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.1 : INTRODUCING SOME MORE ASSEMBLER DIRECTIVES

One of the uses of shift operators is for initializing the registers. For example suppose we want to set the Z and C bits of the SREG (Status Register) register and clear the others. Look at Figure 6-1.

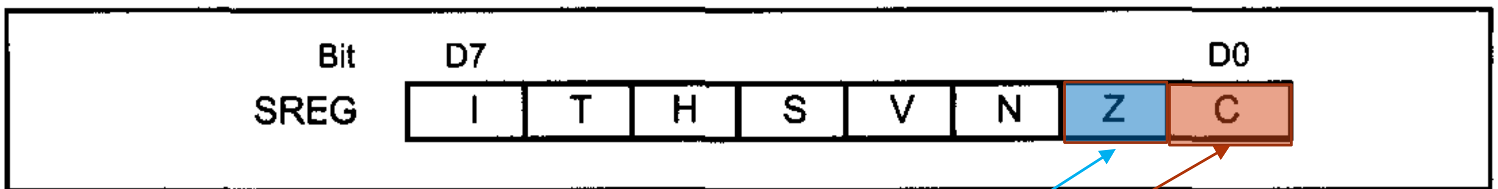


Figure 6-1. Bits of the Status Register

If we load 0b00000011 to SREG the task will be done:

```
LDI R20, 0b00000011   ;Z = 1, C = 1
OUT SREG,R20
```

In this example, we calculated the 0b00000011 number by looking at Figure 6-1.

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.1 : INTRODUCING SOME MORE ASSEMBLER DIRECTIVES

But imagine you are writing a program and you want to do the same task; you have to open the datasheet or a reference book to see the structure of the SREG register. To make the task simpler, the names of the register bits are defined in the header files of each AVR microcontroller. For example, in M32DEF.INC there are the following lines of code:

```
...
; SREG - Status Register
.equ  SREG_C      = 0   ;carry flag
.equ  SREG_Z      = 1   ;zero flag
.equ  SREG_N      = 2   ;negative flag
.equ  SREG_V      = 3   ;2's complement overflow flag
.equ  SREG_S      = 4   ;sign bit
.equ  SREG_H      = 5   ;half carry flag
.equ  SREG_T      = 6   ;bit copy storage
.equ  SREG_I      = 7   ;global interrupt enable
...
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.1 : INTRODUCING SOME MORE ASSEMBLER DIRECTIVES

So, we can use the names of the bits instead of remembering the structure of the registers or finding them in the datasheet. For example, the following program sets the Z flag of the SREG register and clears the other bits:

```
LDI   R16, 1<<SREG_Z ;R16= 1 << 1 = 0b00000010
OUT   SREG,R16       ;SREG = 0b00000010 (set Z and clear others)
```

As another example, the following program sets the V and S flags of SREG:

```
LDI   R16,(1<<SREG_V)|(1<<SREG_S)      ;R16=0b1000|0b10000=0b11000
OUT   SREG,R16    ;SREG = 0b00011000 (set V and S, clear others)
```

In Example 6-1, you see the usage of the directives in I/O port programming.

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.1 : INTRODUCING SOME MORE ASSEMBLER DIRECTIVES

Example 6-1

Write codes to set PB2 and PB4 of PORTB to 1 and clear the other pins
(a) without the directives, and
(b) using the directives.

Solution:

(a)

```
LDI    R20,0x14             ;R20 = 0x14
OUT    PORTB,R20            ;PORTB = R20
```

To make the code more readable, we can write the number in binary as well:

```
LDI    R20,0b00010100       ;R20 = 0x14
OUT    PORTB,R20            ;PORTB = 0x14
```

(b)

```
LDI    R20,(1<<4)|(1<<2) ;R20 = (0b10000 | 0b00100) = 0b10100
OUT    PORTB,R20            ;PORTB = R20
```

As we mentioned before, the names of the register bits are defined in the header files of each AVR microcontroller. PB2 and PB4 are defined equal to 2 and 4, as well. Therefore, we can write the code as shown below:

```
LDI    R20,(1<<PB4)|(1<<PB2) ;set the PB4 and PB2 bits
OUT    PORTB,R20                ;PORTB = R20
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.1 : INTRODUCING SOME MORE ASSEMBLER DIRECTIVES

Notice that when the assembler wants to convert a code to machine language it substitutes all of the assembler directives with their equivalent values. Thus, using the directives has no side effects on the performance of our code but rather makes our code more readable.

Example 6-2

What does the AVR assembler do while assembling the following program?

```
.equ  C1 = 2
.equ  C2 = 3
LDI R20,C1|(1<<C2) ;R20= 2|(1<<3)= 0b00000010|0b00001000= 0b00001010
```

Solution:

.equ is an assembler directive. When assembling ".equ C1 = 2", the assembler assigns value 2 to C1. Similarly, while assembling the ".equ C2 = 3" instruction, it assigns the value 3 to C2.

When the assembler converts the "`LDI  R20,C1|(1<<C2)`" instruction to machine language, it knows the values of C1 and C2. Thus it calculates the value of "`C1|(1<<C2)`", and then replaces the expression with its value. Therefore, "`LDI R20,C1|(1<<C2)`" will be converted to "`LDI  R20,0b00001010`". Then the assembler converts the instruction to machine language.

6.1 : INTRODUCING SOME MORE ASSEMBLER DIRECTIVES

Example 6-3

What does the AVR assembler do while assembling the following program?

```
.INCLUDE "M32DEF.INC"
      LDI   R20,(1<<PB4)|(1<<PB2);set the PB4 and PB2 bits
      OUT   DDRB,R20          ;DDRB = R20
HERE: RJMP  HERE
```

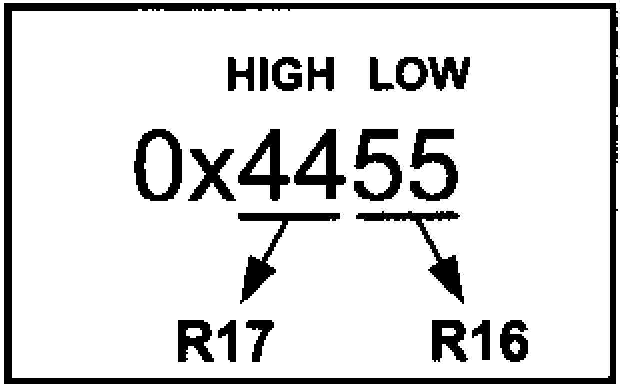
Solution:

Including a header file at the beginning of a program is similar to copying all the contents of the header file to the beginning of the program. Thus, the assembler, first assembles the contents of M32DEF.INC. The header file contains some ".equ" instructions, such as "`.equ PB4 = 4`". Thus, after reading the header file the assembler learns that PB4 is equal to 4, PB2 is equal to 2, and so on. Thus, when it wants to assemble instructions such as "`LDI R20,(1<<PB4)|(1<<PB2)`", it knows the values of PB2 and PB4. It calculates the value of "`(1<<PB4)|(1<<PB2)`" and substitutes it.

HIGH( ) and LOW( ) functions

The HIGH( ) and LOW( ) functions give the higher and the lower bytes of a 16-bit value. For example, in the following program 0x55 and 0x44 are loaded into R16 and R17, respectively:

```
LDI   R16,LOW(0x4455)  ;R16 = 0x55
LDI   R17,HIGH(0x4455) ;R17 = 0x44
```



In Chapter 2, we used the following instructions to make the stack pointer refer to the last location of the memory:

```
LDI   R16,HIGH(RAMEND) ;R16 = 0x08 (for ATmega32)
OUT   SPH,R16          ;SPH = the high byte of address
LDI   R16,LOW(RAMEND)  ;R16 = 0x5f
OUT   SPL,R16          ;SPL = the low byte of address
```

REGISTER AND DIRECT ADDRESSING MODES

The CPU can access data in various ways. The data could be in a register, or in memory, or provided as an immediate value. These various ways of accessing data are called ***addressing modes.*** These ways can be categorized into the following groups:

1. Single-Register (Immediate)
2. Register
3. Direct
4. Register indirect
5. Flash Direct
6. Flash Indirect

Single-register (immediate) addressing mode

In this addressing mode, the operand is a register.

```
NEG   R18              ;negate the contents of R18
COM   R19              ;complement the contents of R19
INC   R20              ;increment R20
DEC   R21              ;decrement R21
ROR   R22              ;rotate right R22
```

In some of the instructions there is also a constant value with the register operand.

```
LDI   R19,0x25         ;load 0x25 into R19
SUBI  R19,0x6          ;subtract 0x6 from R19
ANDI  R19,0b01000000   ;AND R19 with 0x40
```

The constant value is sometimes referred to as ***immediate data*** since the operand comes immediately after the opcode when the instruction is assembled; and the addressing mode is referred to as ***immediate addressing mode*** in some microcontrollers.

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.1 : INTRODUCING SOME MORE ASSEMBLER DIRECTIVES

This addressing mode can be used to load data into any of the R16-R31 general purpose registers. The immediate addressing mode is also used for arithmetic and logic instructions. Note that the letter "I" in instructions such as LDI, ANDI, and SUBI means "Immediate."

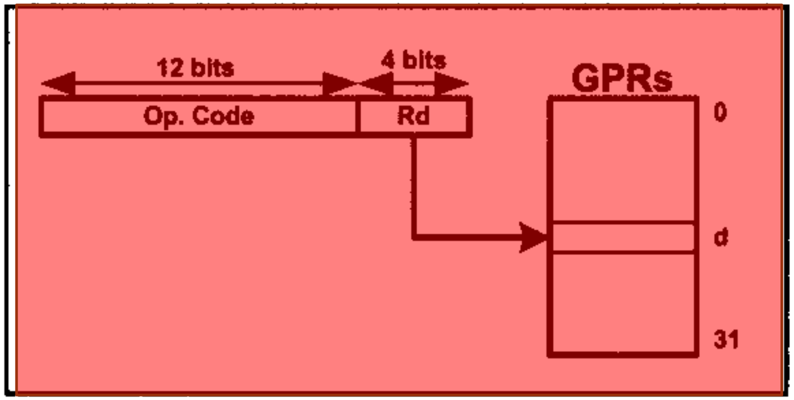


Figure 6-2a. Single-Register Addressing

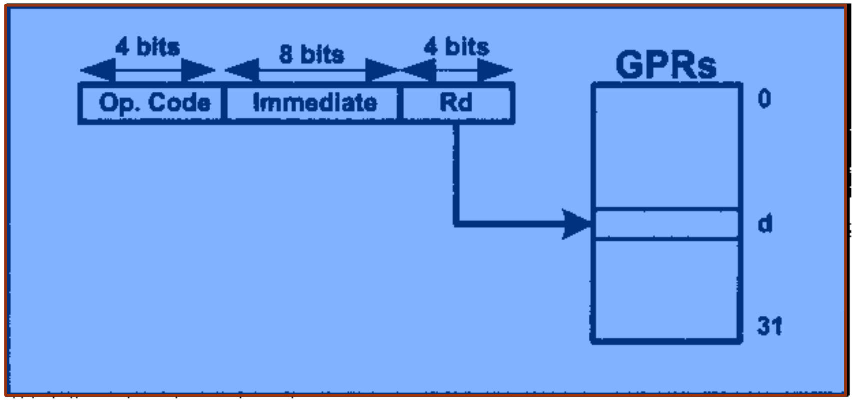


Figure 6-2b. Single-Register (with immediate)

We can use the .EQU directive to access immediate data, as shown below.

```
.EQU COUNT = 0x30
...     ..
LDI   R16,COUNT          ;R16 = 0x30
```

Two-register addressing mode

Two-register addressing mode involves the use of two registers to hold the data to be manipulated.

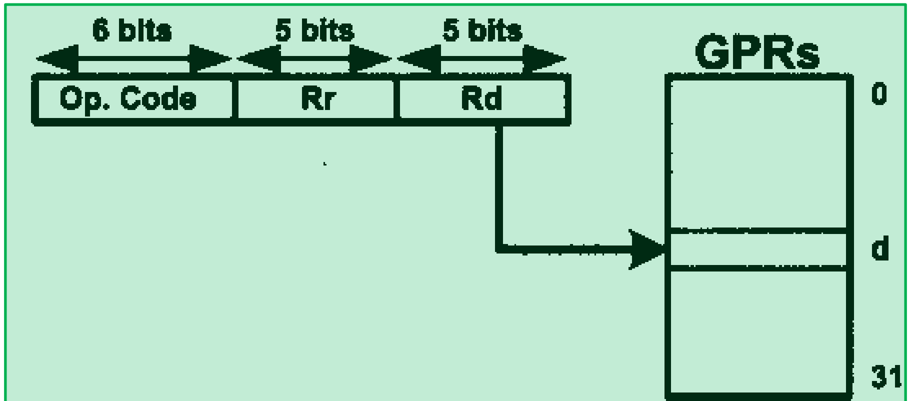


Figure 6-3. Two-Register Addressing

Examples of two-register addressing mode are as follows:

```
ADD   R20,R23          ;add R23 to R20
SUB   R29,R20          ;subtract R20 from R29
AND   R16,R17          ;AND R16 with 0x40
MOV   R23,R19          ;copy the contents of R19 to R23
```

Direct addressing mode

The entire data memory can be accessed using either direct or register indirect addressing modes. In direct addressing mode, the operand data is in a RAM memory location whose address is known, and this address is given as a part of the instruction.

```
LDS    R19,0x560  ;load R19 with the contents of memory loc $560
STS    0x40,R19    ;store R19 to data space location 0x40
```

The two instructions use direct addressing mode. If we dissect the opcode we see that the addresses are embedded in The instruction

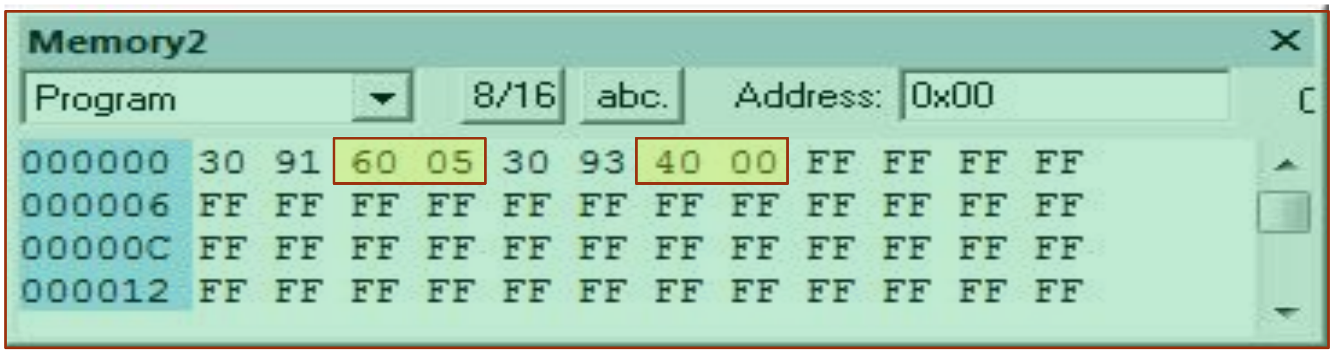


Figure 6-4. Direct Addressing Opcode

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.1 : INTRODUCING SOME MORE ASSEMBLER DIRECTIVES

As shown in Figure 6-5a, the address field is a 16-bit address and can take values from \$0000-\$FFFF. Of course, it is much easier to use names instead of addresses in the program, and we have seen many examples of them in the last few chapters.

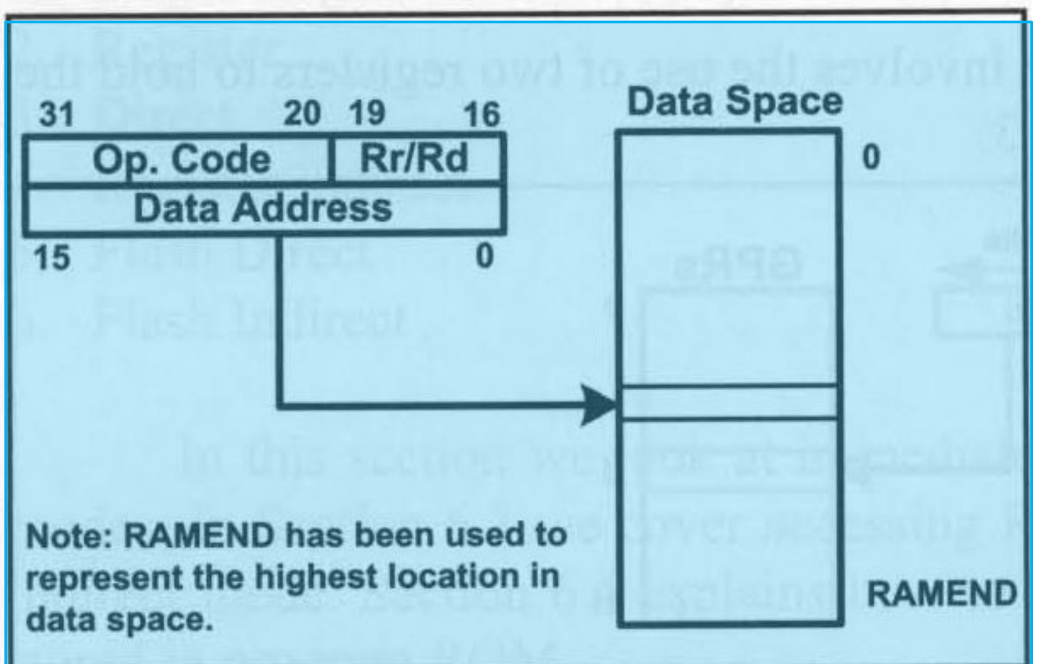


Figure 6-5a. Direct Data Addressing

It must be noted that data memory does not support immediate addressing mode. In other words, to move data into internal RAM or to I/O registers, we must first move it to a GPR (R16-R31), and then move it from the GPR to the data memory space using the STS instruction.

```
LDI     R19,0x95
STS     0x520,R19
```

I/O direct addressing mode

To access the I/O registers there is a special mode called I/O direct addressing mode. The I/O direct addressing mode can address only the standard I/O registers. The IN and OUT instructions use this addressing mode. Examine the following instruction, which copies the contents of PINB to PORTC:

```
IN      R18,0x16        ;PINB
OUT     0x15,R18        ;PORTC
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.1 : INTRODUCING SOME MORE ASSEMBLER DIRECTIVES

As shown in Figure 6-5b, the address field is a 6-bit address and can take values from \$00 to \$3F, which is from 00 to 63 in decimal. So, it can address the entire standard I/O register memory space.

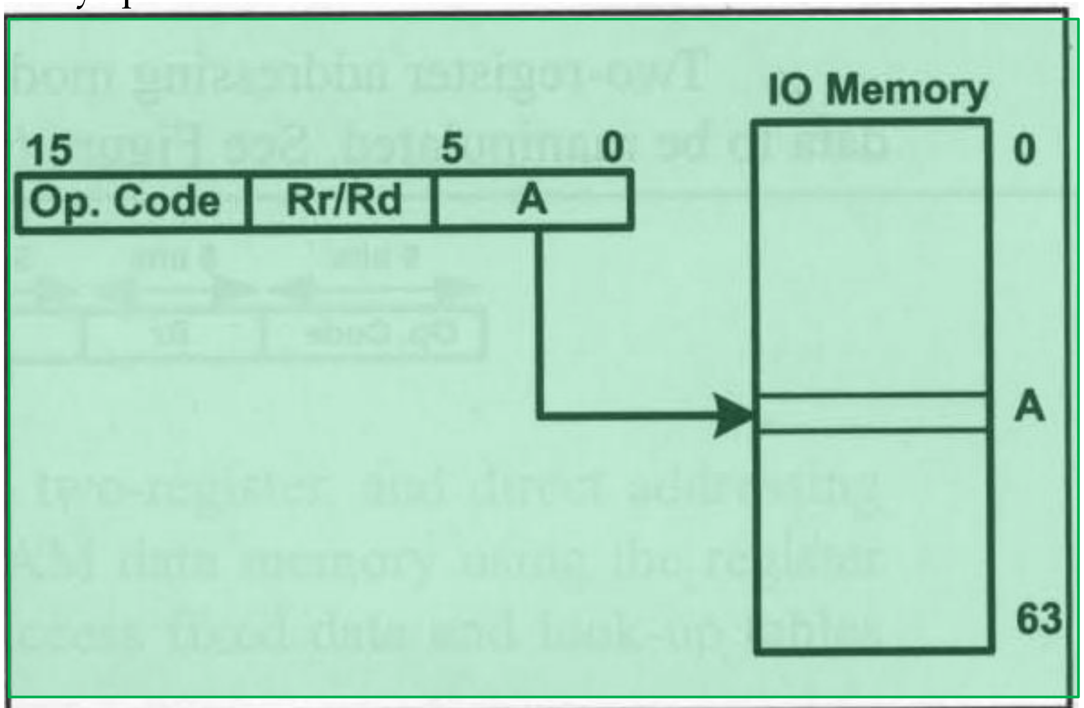


Figure 6-5b. I/O Direct Addressing

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.1 : INTRODUCING SOME MORE ASSEMBLER DIRECTIVES

The AVR registers for Ports A, B, and so on are part of the group of registers commonly referred to as *I/O* ***registers***. There are many I/O registers and they are widely used. The I/O registers can be accessed by their names or by their addresses. For example, PINB has address 0x16, and PORTC the address $15, as shown in Table 6-4. Notice how the following pairs of instructions mean the same thing:

```
OUT     0x15,R19
OUT     PORTC,R19
```

```
IN      R19,0x16
IN      R19,PINB
```


mashhoun@iust.ac.ir Iran Univ of Science & Tech

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING

6.1 : INTRODUCING SOME MORE ASSEMBLER DIRECTIVES

Table 6-4: Selected ATmega32 I/O Register Addresses

Symbol	Name	I/O Address	Data Memory Addr.
PIND	Port D input pins	$10	$30
DDRD	Data Direction, Port D	$11	$31
PORTD	Port D data register	$12	$32
PINC	Port C input pins	$13	$33
DDRC	Data Direction, Port C	$14	$34
PORTC	Port C data register	$15	$35
PINB	Port B input pins	$16	$36
DDRB	Data Direction, Port B	$17	$37
PORTB	Port B data register	$18	$38
PINA	Port A input pins	$19	$39
DDRA	Data Direction, Port A	$1A	$3A
PORTA	Port A data register	$1B	$3B
SPL	Stack Pointer, Low byte	$3D	$5D
SPH	Stack Pointer, High byte	$3E	$5E

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.1 : INTRODUCING SOME MORE ASSEMBLER DIRECTIVES

Table 6-4 lists some of the AVR I/O registers and their addresses. The following points should be noted about the addresses of I/O registers:

1. As shown in Figures 2-3 and **2-4, the addresses between $20 and $5F of the** data space have been assigned to standard I/O registers in all of the AVRs. These I/O registers have two addresses: I/O address and data memory address. The I/O address is used when we use the I/O direct addressing mode, while the data memory address is used when we use the direct addressing mode; in other words, the standard I/O registers can be accessed using both the direct addressing and the I/O addressing modes.
2. Some AVRs have less than 64 I/O registers. So, some locations of the standard I/O memory are not used by the I/O registers. The unused locations are reserved and must not be used by the AVR programmer.

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING

6.1 : INTRODUCING SOME MORE ASSEMBLER DIRECTIVES

3. Some AVRs have more than 64 I/O registers. The extra I/O registers are located above the data memory address $5F. The data memory allocated to the extra I/O registers is called extended I/O memory.
   To access the extended I/O registers we can use the direct addressing mode. For example, inATmega128, PORTF has the memory address of 0x62. So, the following instruction stores the contents of R20 in PORTF.

```
STS      0x62,R20
```

4. The I/O registers can have different addresses in different AVR microcontrollers. For example, the I/O address $2 is assigned to TWAR in the ATmega32, while the same address is assigned to DDRE inATmega128. This can cause problems if you want to run programs written for one AVR on another AVR. The best way to solve this problem is to use the names of the registers instead of their addresses.

Register indirect addressing mode

In the register indirect addressing mode, a register is used as a pointer to the data memory location. In the AVR, three registers are used for this purpose: X, Y, and Z. These are 16-bit registers allowing access to the entire 65,536 bytes of data memory space in the AVR.

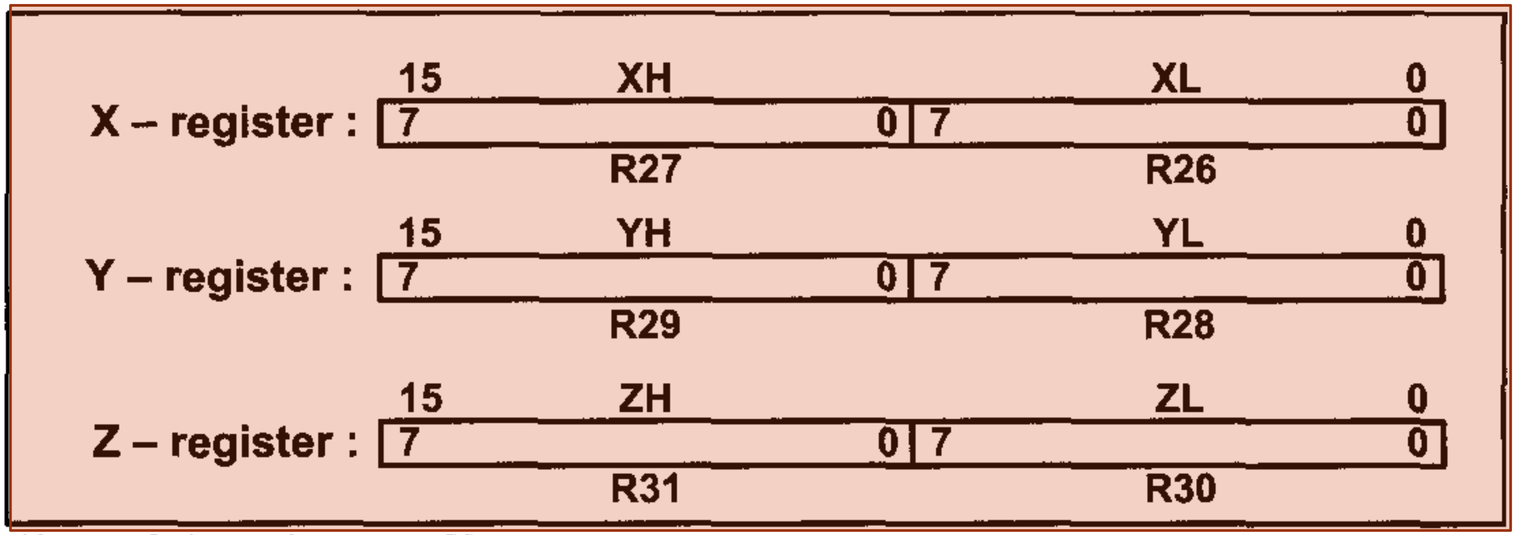


Figure 6-6. Registers X, Y, and Z

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.3 REGISTER INDIRECT ADDRESSING MODE

We can use them with the LD instruction to read the value of a location pointed to by these registers. For example, the following instruction reads the value of the location pointed to by the X pointer.

```
LD    R24, X        ;load into R24 from location pointed to by X
```

For instance, the following program loads the contents of location 0x130 into R18:

```
LDI   XL, 0x30      ;load R26 (the low byte of X) with 0x30
LDI   XH, 0x01      ;load R27 (the high byte of X) with 0x1
LD    R18, X        ;copy the contents of location 0x130 to R18
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.3 REGISTER INDIRECT ADDRESSING MODE

The ST instruction can be used to write a value to a location to which any of the X, Y, and Z registers points. For example, the following program stores the contents of R23 into location 0x139F:

```
LDI   ZL,0x9F
LDI   ZH,0x13
ST    Z,R23
```

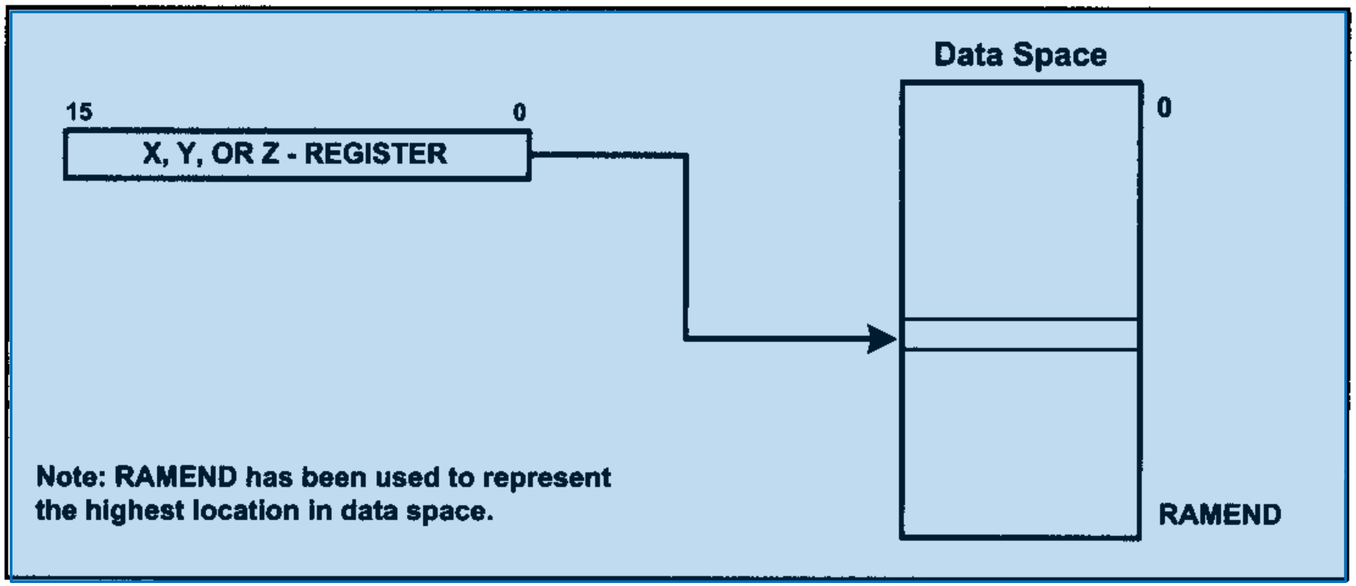


Figure 6-7. Register Indirect Addressing Mode

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.3 REGISTER INDIRECT ADDRESSING MODE

Advantages of register indirect addressing mode

C ... at it makes

acces ... mode.

Example 6-5

Write a program to copy the value $55 into memory locations $140 through $144 using
(a) direct addressing mode,
(b) register indirect addressing mode without a loop, and
(c) a loop.

Solution:

(a)
```
      LDI   R17,0x55          ;load R17 with value 0x55
      STS   0x140,R17         ;copy R17 to memory location 0x140
      STS   0x141,R17         ;copy R17 to memory location 0x141
      STS   0x142,R17         ;copy R17 to memory location 0x142
      STS   0x143,R17         ;copy R17 to memory location 0x143
      STS   0x144,R17         ;copy R17 to memory location 0x144
```

(b)
```
      LDI   R16,0x55    ;load R16 with value 0x55
      LDI   YL,0x40     ;load R28 with value 0x40 (low byte of addr.)
      LDI   YH,0x1      ;load R29 with value 0x1 (high byte of addr.)
      ST    Y,R16       ;copy R16 to memory location 0x140
      INC   YL          ;increment the low byte of Y
      ST    Y,R16       ;copy R16 to memory location 0x141
      INC   YL          ;increment the pointer
      ST    Y,R16       ;copy R16 to memory location 0x142
      INC   YL          ;increment the pointer
      ST    Y,R16       ;copy R16 to memory location 0x143
      INC   YL          ;increment the pointer
      ST    Y,R16       ;copy R16 to memory location 0x144
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.3 REGISTER INDIRECT ADDRESSING MODE

Example 6-5 (Cont.)

```
(c)    LDI    R16,0x5      ;R16 = 5 (R16 for counter)
       LDI    R20,0x55     ;load R20 with value 0x55 (value to be copied)
       LDI    YL,0x40      ;load YL with value 0x40
       LDI    YH,0x1       ;load YH with value 0x1
L1:    ST     Y,R20        ;copy R20 to memory pointed to by Y
       INC    YL           ;increment the pointer
       DEC    R16          ;decrement the counter
       BRNE   L1           ;loop while counter is not zero
```

Use the AVR Studio simulator to examine memory contents after the above program is run.

$140 = ($55) $141 = ($55) $142 = ($55) $143 = ($55)   144 = ($55)

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.3 REGISTER INDIRECT ADDRESSING MODE

Auto-increment and auto-decrement options for pointer registers

Because the pointer registers (X, Y, and Z) are 16-bit registers, they can go from \$0000 to \$FFFF, which covers the entire 64K memory space of the AVR. Using the "INC ZL" instruction to increment the pointer can cause a problem.

Table 6-5: AVR Auto-Increment/Decrement of Pointer Registers for LD Instruction

Instruction		Function
LD	Rn,X	After loading location pointed to by X, the X stays the same.
LD	Rn,X+	After loading location pointed to by X, the X is incremented.
LD	Rn,-X	The X is decremented, then the location pointed to by X is loaded.
LD	Rn,Y	After loading location pointed to by Y, the Y stays the same.
LD	Rn,Y+	After loading location pointed to by Y, the Y is incremented.
LD	Rn,-Y	The Y is decremented, then the location pointed to by Y is loaded.
LDD	Rn,Y+q	After loading location pointed to by Y+q, the Y stays the same.
LD	Rn,Z	After loading location pointed to by Z, the Z stays the same.
LD	Rn,Z+	After loading location pointed to by Z, the Z is incremented.
LD	Rn,-Z	The Z is decremented, then the location pointed to by Z is loaded.
LDD	Rn,Z+q	After loading location pointed to by Z+q, the Z stays the same.

Example 6-6

Assume that RAM location \$90-\$94 have string of ASCII data, as shown below:

\$90 = ('H') \$91 = ('E') \$92 = ('L') \$93 = ('L') \$94 = ('O')

Write a program to get each character and send it to PORT B one byte at a time. Show the program using :

(a) Direct Accessing mode.

(b) Register indirect addressing mode.

Solution:

(a) Using direct Addressing mode

```
LDI     R20,0xFF
OUT     DDRB,R20
LDS     R20,0x90
OUT     PORTB,R20
LDS     R20,0x91
OUT     PORTB,R20
LDS     R20,0x92
OUT     PORTB,R20
LDS     R20,0x93
OUT     PORTB,R20
LDS     R20,0x94
OUT     PORTB,R20
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.3 REGISTER INDIRECT ADDRESSING MODE

(b) Using Register indirect addressing mode

```
        LDI     R16,0x5
        LDI     R20,0xFF
        OUT     DDRB,R20
        LDI     ZL,0x90
        LDI     ZH,0x0
L1:     LD      R20,Z
        INC     ZL
        OUT     PORTB,R20
        DEC     R16
        BRNE    L1
```

When simulating the above program on the AVR Studio, make sure memory location \$90-\$94 have the message "HELLO"

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.3 REGISTER INDIRECT ADDRESSING MODE

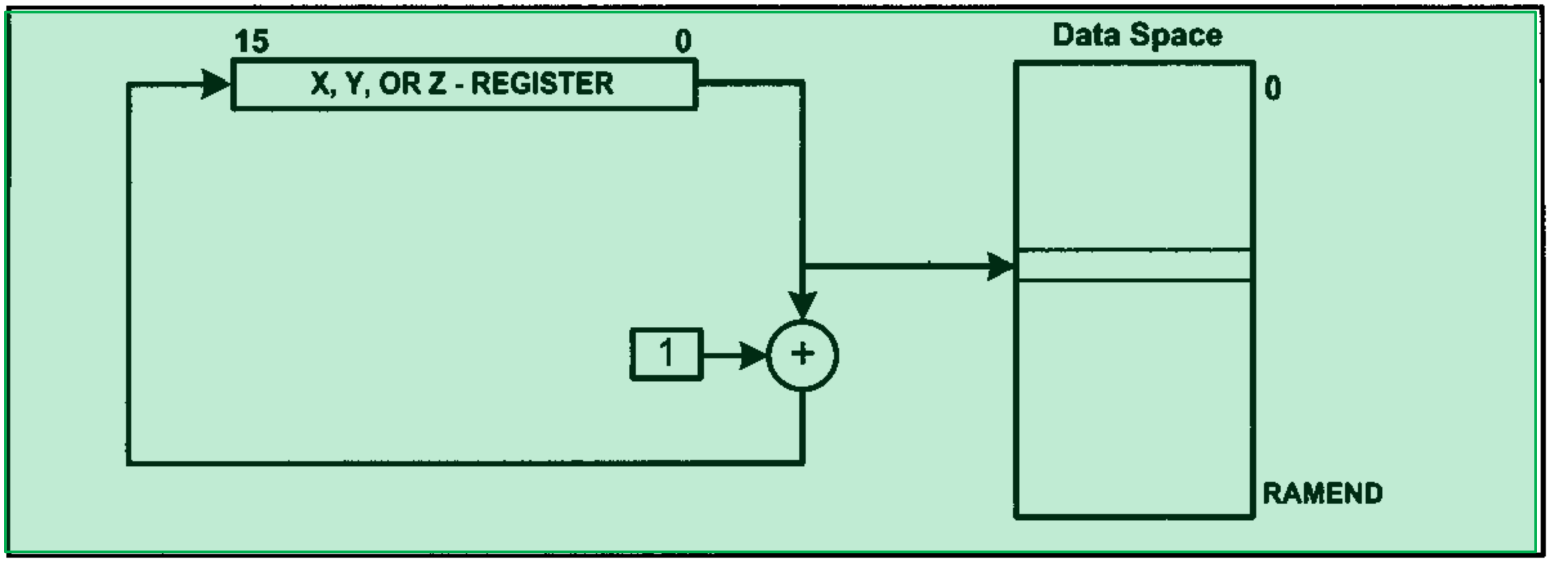


Figure 6-8. Register Indirect Addressing with Post-increment

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.3 REGISTER INDIRECT ADDRESSING MODE

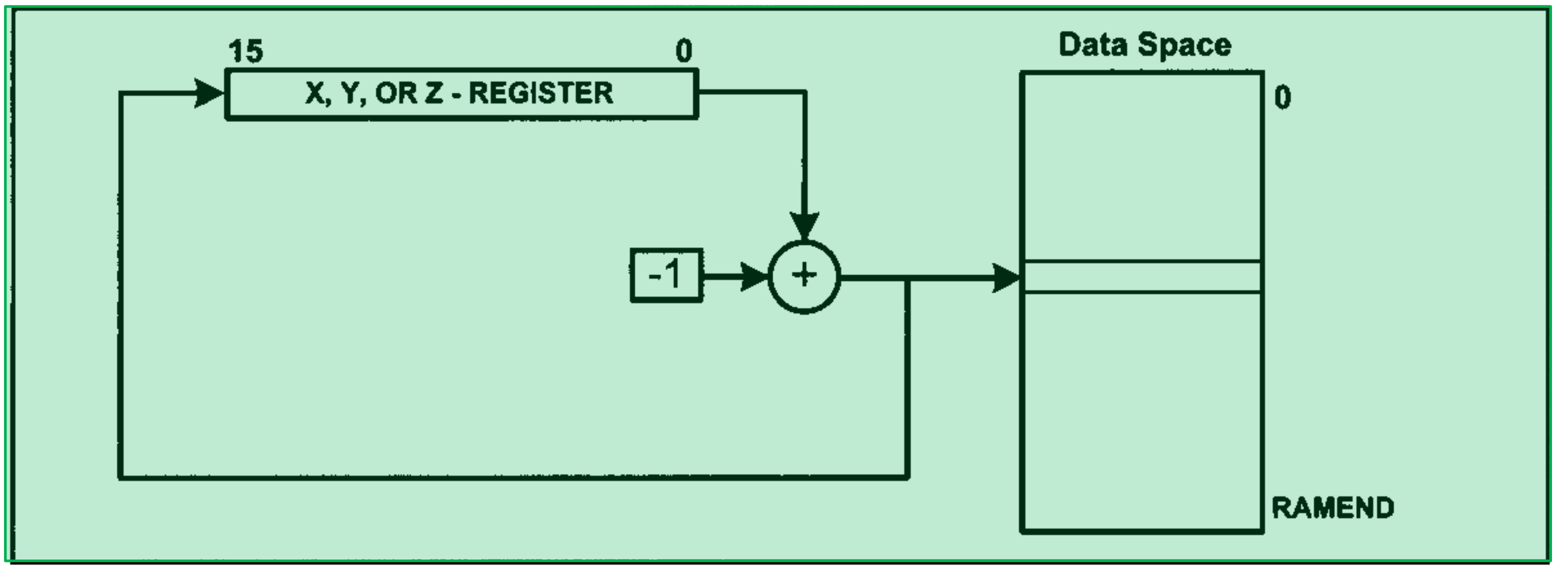


Figure 6-9. Register Indirect Addressing with Pre-decrement

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.3 REGISTER INDIRECT ADDRESSING MODE

Write a program to clear 16 memory location starting at data memory address \$60. Use the following:

(a) INC Rn

(b) Auto-increment

```
.include "m32def.inc"
        LDI     R16,16
        LDI     XL,0x60
        LDI     XH,0x00
        LDI     R20,0x0
L1:     ST      X,R20
        INC     XL
        DEC     R16
        BRNE    L1
```

```
.INCLUDE "m32def.inc"
        LDI     R16,16
        LDI     XL,0x60
        LDI     XH,0x00
        LDI     R20,0x0
L1:     ST      X+,R20
        DEC     R16
        BRNE    L1
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.3 REGISTER INDIRECT ADDRESSING MODE

Example 6-8

Assume that data memory locations $240–$243 have the following hex data. Write a program to add them together and place the result in locations $220 and $221.

$240 = ($7D) $241 = ($EB) $242 = ($C5) $243 = ($5B)

Solution:

```
        .INCLUDE "M32DEF.INC"
        .EQU L_BYTE = 0x220     ;RAM loc for L_Byte
        .EQU H_BYTE = 0x221     ;RAM loc for H_Byte
        LDI   R16,4
        LDI   R20,0
        LDI   R21,0
        LDI   XL, 0x40     ;the low byte of X = 0x40
        LDI   XH, 0x02     ;the high byte of X = 02
L1:     LD    R22, X+      ;read contents of location where X points to
        ADD   R20, R22
        BRCC  L2           ;branch if C = 0
        INC   R21          ;increment R21
L2:     DEC   R16          ;decrement counter
        BRNE  L1           ;loop until counter is zero
        ST    L_BYTE, R20 ;store the low byte of the result in $220
        ST    H_BYTE, R21 ;store the high byte of the result in $221
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.3 REGISTER INDIRECT ADDRESSING MODE

Example 6-9

Write a program to copy a block of 5 bytes of data from data memory locations starting at \$130 to RAM locations starting at \$60.

Solution:

```
        LDI     R16, 16         ;R16 = 16 (counter value)
        LDI     XL, 0x30        ;the low byte of address
        LDI     XH, 0x01        ;the high byte of address
        LDI     YL, 0x60        ;the low byte of address
        LDI     YH, 0x00        ;the high byte of address
L1:     LD      R20, X+         ;read where X points to
        ST      Y+, R20         ;store R20 where Y points to
        DEC     R16             ;decrement counter
        BRNE    L1              ;loop until counter = zero
```

Before we run the above program.

```
130 = ('H') 131 = ('E') 132 = ('L') 133 = ('L') 134 = ('O')
```

After the program is run, the addresses \$60–\$64 have the same data as \$130–\$134.

```
130 = ('H') 131 = ('E') 132 = ('L') 133 = ('L') 134 = ('O')
60 = ('H')  61 = ('E')  62 = ('L')  63 = ('L')  64 = ('O')
```


mashhoun@iust.ac.ir Iran Univ of Science & Tech

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.3 REGISTER INDIRECT ADDRESSING MODE

Example 6-10

Two multibyte numbers are stored in locations \$130–\$133 and \$150–\$153. Write a program to add the multibyte numbers and save the result in address \$160–\$163.

```
        $C7659812
+       $2978742A
```

Solution:

```
        .INCLUDE "M32DEF.INC"
        LDI     R16, 4          ;R16 = 4 (counter value)
        LDI     XL, 0x30
        LDI     XH, 0x1         ;load pointer. X = $130
        LDI     YL, 0x50
        LDI     YH, 0x1         ;load pointer. Y = $150
        LDI     ZL, 0x60
        LDI     ZH, 0x1         ;load pointer. Z = $160
        CLC                     ;clear carry
L1:     LD      R18, X+         ;copy memory to R18 and INC X
        LD      R19, Y+         ;copy memory to R19 and INC Y
        ADC     R18,R19         ;R18 = R18 + R19 + carry
        ST      Z+,R18          ;store R18 in memory and INC Z
        DEC     R16             ;decrement R16 (counter)
        BRNE    L1              ;loop until counter = zero
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.3 REGISTER INDIRECT ADDRESSING MODE

Before the addition we have:

```
     MSByte                                          LSByte
     133 = ($C7) 132 = ($65) 131 = ($98)  130 = ($12)
     153 = ($29) 152 = ($78) 151 = ($74)  150 = ($2A)
```

After the addition we have:

```
     163 = ($F0)  162 = ($DE) 161 = (0C)  160 = (3C)
```

Notice that we are using the little endian convention of storing a low byte to a low address, and a high byte to a high address. Single-step the program in AVR Studio and examine the pointer registers and memory contents to gain insight into register indirect addressing mode.

Register indirect with displacement

Suppose we want to read a byte that is a few bytes higher than where the Z register points to. To do so we can increment the Z register so that it points to the desired location and then read it. But there is an easier way; we can use the register indirect with displacement. In this addressing mode a fixed number is added to the Z register. For example, if we want to read from the location that is 5 bytes after the location to which Z points, we can write the following instruction:

```
LDD    R20, Z+5       ;load from Z+5 into R20
```

The general format of the instruction is as follows:

```
LDD    Rd, Z+q        ;load from Z+q into Rd
```

where q is a number between 0 to 63, and Rd is any of the general purpose registers.

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.3 REGISTER INDIRECT ADDRESSING MODE

To store a byte of data in a data memory location using the register indirect with displacement addressing mode we can use STD (Store with Displacement). The instruction is as follows:

```
STD    Z+q,Rr        ;store Rr into location Z+q
```

For example, the following instruction writes the contents of R20 into the location that is five bytes away from where Z points to:

```
STD    Z+5,R20       ;store R20 into location Z+5
```


mashhoun@iust.ac.ir

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.3 REGISTER INDIRECT ADDRESSING MODE

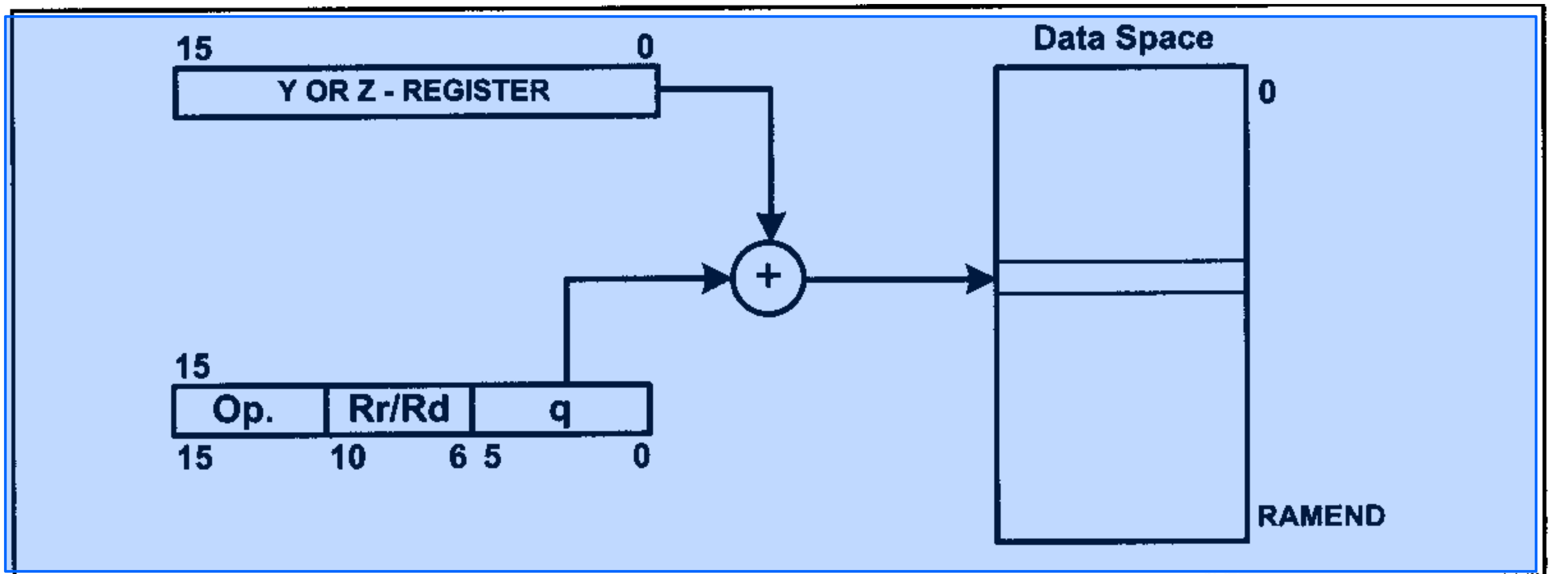


Figure 6-10. Register Indirect with Displacement

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.3 REGISTER INDIRECT ADDRESSING MODE

Example 6-11

Write a function that adds the contents of three continuous locations of data space and stores the result in the first location. The Z register should point to the first location before the function is called.

Solution:

```
.INCLUDE "M32DEF.INC"
        LDI    R16,HIGH(RAMEND)   ;initialize the stack pointer
        OUT    SPH,R16
        LDI    R16,LOW(RAMEND)
        OUT    SPL,R16
        LDI    ZL,0x00            ;initialize the Z register
        LDI    ZH,2
        CALL   ADD3LOC            ;call add3loc
HERE:   JMP    HERE               ;loop forever
ADD3LOC:
        LDI    R21,0              ;R21 = 0
        LD     R20,Z              ;R20 = contents of location  Z
        LDD    R16,Z+1            ;R16 = contents of location Z+1
        ADD    R20,R16            ;R20 = R20 + R16
        BRCC   L1                 ;branch if carry cleared
        INC    R21                ;increment R21 if carry occurred
L1:     LDD    R16,Z+2            ;R16 = contents of location Z+2
        ADD    R20,R16            ;R20 = R20 + R16
        BRCC   L2                 ;branch if carry cleared
        INC    R21                ;increment R21
L2:     ST     Z,R20              ;store R20 into location Z
        STD    Z+1,R21            ;store R21 into location Z+1
        RET
```


mashhoun@iust.ac.ir

.DB (define byte) and fixed data in program ROM

The **.DB** data directive is widely used to allocate ROM program (code) memory in byte-sized chunks. When **.DB** is used to define fixed data, the numbers can be in decimal, binary, hex, or ASCII formats. The **.DB** directive is widely used to define ASCII strings.

Example 6-12

Assume that we have burned the following fixed data into the program ROM of an AVR chip. Give the contents of each ROM location starting at $500. See Appendix F for the hex values of the ASCII characters.

```
;MY DATA IN FLASH ROM
      .ORG $500
DATA1: .DB 1,8,5,3
DATA2: .DB 28          ;DECIMAL(1C in hex)
DATA3: .DB 0b00110101  ;BINARY (35 in hex)
DATA4: .DB 0x39        ;HEX
      .ORG 0x510
DATA4: .DB 'Y'         ;single ASCII char
DATA5: .DB '2','0','0','5';ASCII numbers
      .ORG $516
DATA6: .DB "Hello ALI"  ;ASCII string
```

.DB (define byte) and fixed data in program ROM

In Example 6-12 notice that each location of program memory is 2 bytes, whereas the . DB directive allocates byte-sized chunks. If we allocate a few bytes of data using the . DB directive, the first byte goes to the low byte of ROM location; the second byte goes to the high byte of ROM location; the third byte goes to the low byte of the next location of program ROM; and so on.

In the cases in which we allocate an odd number of ROM locations using . DB, the assembler will automatically make the number of allocated locations even by placing a zero into the high byte of the last location.

In Example 6-12 notice also that we must use single quotes (') for a single character and double quotes (") for a string.

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.4 LOOK-UP TABLE AND TABLE PROCESSING

AVR assembly also allows the use of .DW in place of . DB to define values greater than 255 (0xFF) but not larger than 65,535 (0xFFFF).

Example 6-13

Give the contents of each ROM location starting at $600.

```
      .ORG $600
DATA1: .DW 0x1234,0x1122
DATA2: .DW 28           ;DECIMAL (001C in hex)
DATA3: .DW 0x2239       ;HEX
```

Solution:

Since AVR is little endian, the low byte of 0x1234, which is 0x34, goes to the low byte of location $600, and its high byte goes to the high byte.

```
Memory
Program   8/16  abc.  Address: 0x600   Cols: Auto
000600 34 12 22 11 1C 00 39 22 FF FF FF FF FF FF FF FF  4."...9"ÿÿÿÿÿÿÿÿ
000608 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
```

Reading table elements in the AVR

Now, we need to have a register pointer to point to the data to be fetched from the code space. The Z register can be used for this purpose. For this reason we can call it register indirect flash addressing mode.

In AVR terminology, there are two register indirect flash addressing modes:

- *program memory constant addressing*

  and
- *program memory addressing with post-increment.*

In the program memory constant addressing mode, the content of Z does not change when the instruction is executed, which is why is called constant addressing; whereas in the program memory addressing with post-increment, the content of Z increments after each execution.

mashhoun@iust.ac.ir Iran Univ of Science & Tech

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.4 LOOK-UP TABLE AND TABLE PROCESSING

The Instruction

"LPM Rn,Z"

uses program memory constant addressing mode, while

"LPM Rn,Z+"

uses program memory addressing with post-increment.

There is a group of AVR instructions designed for table processing. Table 6.6 shows the instructions for table reading in the AVR.

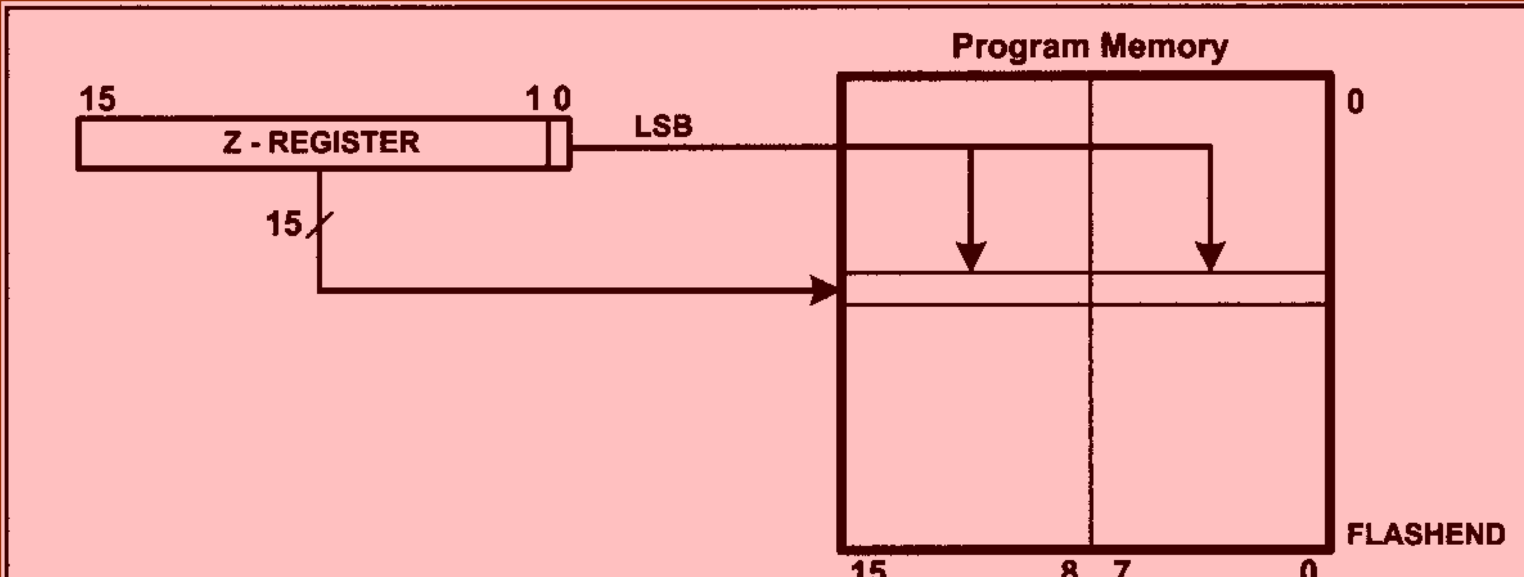
Table 6-6: AVR Table Read Instructions

Instruction	Function	Description
LPM Rn,Z	Load from Program Memory	After read, Z stays the same
LPM Rn,Z+	Load from Program Memory with post-inc.	Reads and increments Z

Note: The byte of data is read into the Rn register from code space pointed to by Z.

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.4 LOOK-UP TABLE AND TABLE PROCESSING

As you know, in the AVR, each location of the program memory is 2 bytes. So, we should mention if we want to read the low byte or the high byte. The least significant bit (LSB) of the Z register indicates whether the low byte or the high byte should be read.



Note: If LSB = 0, the low byte is selected; if LSB = 1, the high byte is selected. Bits 15 through 1 are for word address.

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.4 LOOK-UP TABLE AND TABLE PROCESSING

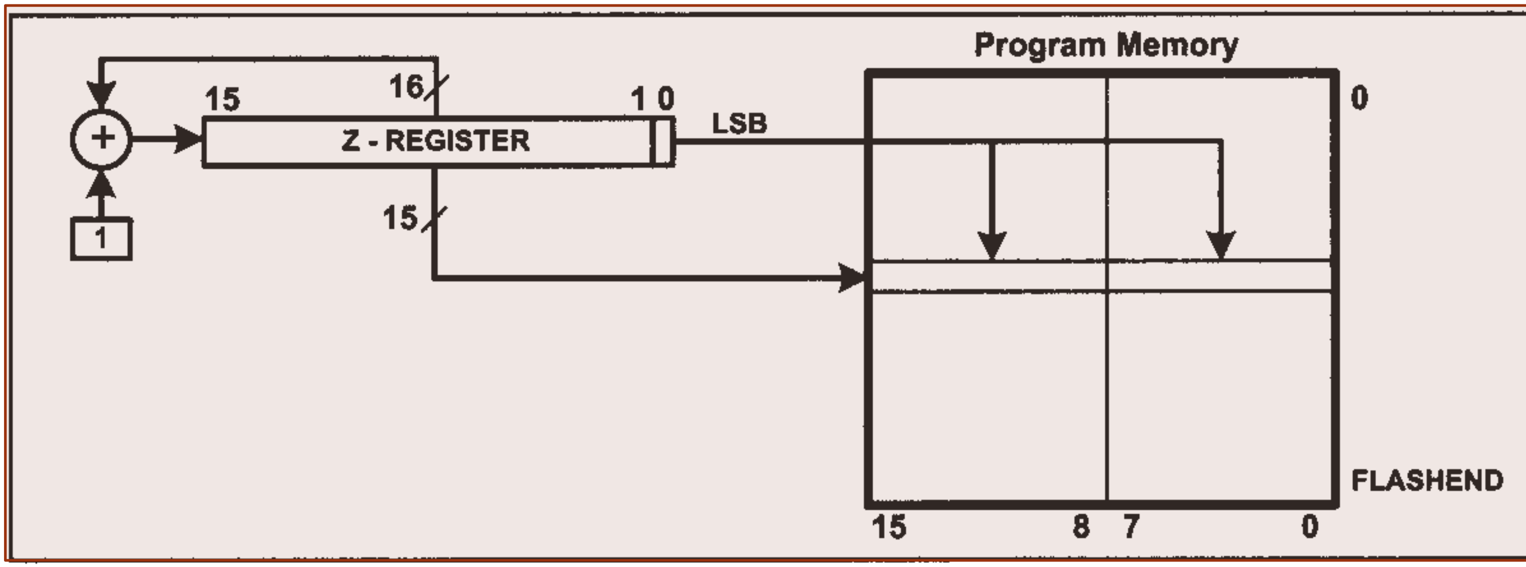


Figure 6-12. Program Memory Addressing with Post-increment

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.4 LOOK-UP TABLE AND TABLE PROCESSING

Figure 6-13b shows the value that should be loaded into the Z register in order to address each byte of the program memory.

For example, to address the low byte of location $0002, we should load the Z register with $0005, as shown below:

```
LDI ZH, 0x00        ;load ZH with 0x00 (the high byte of addr.)
LDI ZL, 0x05        ;load ZL with 0x05 (the low byte of addr.)
LPM R16, Z          ;load R16 with contents of location Z
```

Low	High	Address
0000 0000 0000 0000	0000 0000 0000 0001	0000 0000 0000 0000
0000 0000 0000 0010	0000 0000 0000 0011	0000 0000 0000 0001
0000 0000 0000 0100	0000 0000 0000 0101	0000 0000 0000 0010
0000 0000 0000 0110	0000 0000 0000 0111	0000 0000 0000 0011
0000 0000 0000 1000	0000 0000 0000 1001	0000 0000 0000 0100
0000 0000 0000 1010	0000 0000 0000 1011	0000 0000 0000 0101
⋮	⋮	
1111 1111 1111 1100	1111 1111 1111 1101	0111 1111 1111 1110
1111 1111 1111 1110	1111 1111 1111 1111	0111 1111 1111 1111

Figure 6-13a. Values of Z (in Binary)

Low	High	Address
$0000	$0001	$0000
$0002	$0003	$0001
$0004	$0005	$0002
$0006	$0007	$0003
$0008	$0009	$0004
$000A	$000B	$0005
$FFFC	$FFFD	$7FFE
$FFFE	$FFFF	$7FFF

Figure 6-13b. Values of Z

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.4 LOOK-UP TABLE AND TABLE PROCESSING

As you see in Figure 6-13a, to read the low byte of each location we should shift the address of that location one bit to the left. For instance, to access the low byte of location 0b00000101, we should load Z with 0b000001010.

To read the high byte, we shift the address to the left and we set bit 0 to one.

For example, the following program reads the low byte of location $100:

```
LDI  ZH, HIGH($100<<1) ;load ZH with the high byte of addr.
LDI  ZL, LOW ($100<<1) ;load ZL with the low byte of addr.
LPM R16, Z             ;load R16 with contents of location Z
```

If we OR a number with 1, its bit 0 will be set. Thus, the following program reads the high byte of location $100.

```
LDI  ZH, HIGH(($100<<1)|1)
LDI  ZL, LOW (($100<<1)|1)
LPM R16, Z             ;load R16 with contents of location Z
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING

6.4 LOOK-UP TABLE AND TABLE F

Example 6-14

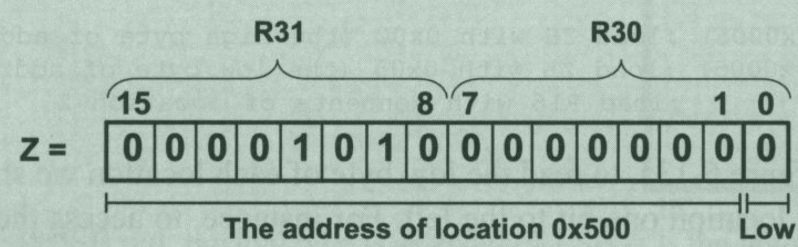
In this program, assume that the phrase “WORLD PEACE” is burned into ROM location starting at $500, and the program is burned into ROM locations starting at 0. Analyze how program works and state where “WORLD PEASE.” is stored after this program is run.

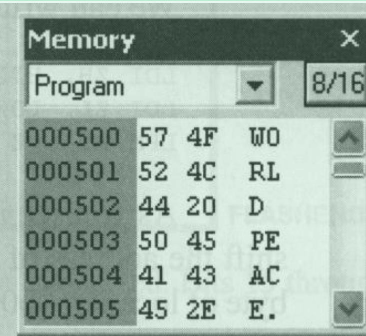
```
.INCLUDE "M32DEF.INC"
        .ORG    $0000
        LDI     R20,0xFF
        OUT     DDRB,R20
        LDI     ZL,LOW(MYDATA<<1)
        LDI     ZH,HIGH(MYDATA<<1)
        LPM     R20,Z
        OUT     PORTB,R20
        INC     ZL
        LPM     R20,Z
        OUT     PORTB,R20
        INC     ZL
        LPM     R20,Z
        OUT     PORTB,R20
        INC     ZL
        LPM     R20,Z
        OUT     PORTB,R20
        INC     ZL
        LPM     R20,Z
        OUT     PORTB,R20
        INC     ZL
        LPM     R20,Z
        OUT     PORTB,R20
        INC     ZL
HERE:   RJMP    HERE

        .ORG    $500
MYDATA: .DB     "WORLD PEACE."
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING

6.4 LOOK-UP TABLE AND TABLE PROCESSING

```
MYDATA: .DB "WORLD PEACE."
```





Solution:

In the above program, ROM locations \$500–\$505 have the following contents.

```
$500 (Low byte) = ('W')        $500 (High byte) = ('O')
$501 (Low byte) = ('R')        $501 (High byte) = ('L')
$502 (Low byte) = ('D')        $502 (High byte) = (' ')
$503 (Low byte) = ('P')        $503 (High byte) = ('E')
$504 (Low byte) = ('A')        $504 (High byte) = ('C')
$505 (Low byte) = ('E')        $505 (High byte) = ('.')
```

We start with Z = \$0A00 (R31:R30 = \$A00). The instruction "`LPM R20,Z`" loads R20 with the contents of the low byte of ROM location \$500. Register R20 contains \$57, the ASCII value for 'W'. This is loaded to Port B. Next, ZL is incremented to make Z = \$A01. The LPM instruction will get the contents of the high byte of ROM location \$500, which is character 'O'. After this program is run, we send the ASCII values for the characters 'W', 'O', 'R', 'L', and 'D' to Port B one character at a time. The loop version of this program is given in the next example.

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING

6.4 LOOK-UP TABLE AND TABLE PROCESSING

Example 6-15

Assuming that program ROM space starting at $500 contains "WORLD PEACE.". Write a program to send characters to Port B one byte at a time.

```
;This method uses a counter
.include "m32def.inc"
         .ORG     $0000
         LDI      R16,11
         LDI      R20,0xFF
         OUT      DDRB,R20
         LDI      ZL,LOW(MYDATA<<1)
         LDI      ZH,HIGH(MYDATA<<1)
L1:      LPM      R20,Z
         OUT      PORTB,R20
         INC      ZL
         DEC      R16
         BRNE     L1
HERE:    RJMP     HERE
;----------------------------------
;data is burned into code (program)
;space starting at $500
         .ORG     $500
MYDATA:  .DB      "WORLD PEASE."
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING

6.4 LOOK-UP TABLE AND TABLE PROCESSING

```
;This method uses a null char for
;end of string
.include "m32def.inc"
        .ORG    $0000
        LDI     R20,0xFF
        OUT     DDRB,R20
        LDI     ZL,LOW(MYDATA<<1)
        LDI     ZH,HIGH(MYDATA<<1)
L1:     LPM     R20,Z
        CPI     R20,0
        BREQ    HERE
        OUT     PORTB,R20
        INC     ZL
        RJMP    L1
HERE:   RJMP    HERE
;---------------------------------
;data is burned into code (program)
;space starting at $500
        .ORG    $500
MYDATA: .DB     "WORLD PEASE.",0
```


mashhoun@iust.ac.ir Iran Univ of Science & Tech

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.4 LOOK-UP TABLE AND TABLE PROCESSING

Auto-increment option for Z

Using the "INC ZL" instruction to increment the pointer can cause a problem when an address such as $5FF is incremented. The carry will not propagate into ZH. The AVR gives us the option of LPM Rn, Z+ (load program memory with post-increment) as shown in Table 6-6.

Example 6-16

```
;burn into ROM starting at 0
.include "m32def.inc"
        .ORG    $0000
        LDI     R20,0xFF
        OUT     DDRB,R20
        LDI     ZL,LOW(MYDATA<<1)
        LDI     ZH,HIGH(MYDATA<<1)
L1:     LPM     R20,Z+
        CPI     R20,0
        BREQ    HERE
        OUT     PORTB,R20
        RJMP    L1
HERE:   RJMP    HERE
;----------------------------------
;burned into code (program) space
;starting at $500
        .ORG    $500
MYDATA: .DB     "WORLD PEASE.",0
```

Example 6-17

Assume that ROM space starting at $100 contains the message "The Promise of World Peace". Write a program to bring this message into the CPU one byte at a time and place the byte in Ram locations starting at $140.

```
.EQU     RAM_BUF = 0X140
.ORG     $0000
.INCLUDE "M32DEF.INC"
         LDI     R20,0xFF
         OUT     DDRB,R20
         LDI     ZH,HIGH(MYDATA<<1)
         LDI     ZL,LOW(MYDATA<<1)
         LDI     XH,HIGH(RAM_BUF)
         LDI     XL,LOW(RAM_BUF)
L1:      LPM     R20,Z+
         CPI     R20,0
         BREQ    HERE
         ST      X+,R20
         RJMP    L1
HERE:    RJMP    HERE
;-------------------------------------
         .ORG    0x100
MYDATA:  .DB     "The Promise of World Peace",0
```

Look-up table

The look-up table is a widely used concept in microcontroller programming. It allows access to elements of a frequently used table with minimum operations. **As an example, assume that for a certain application we need $4+x^2$ values** in the range of 0 to 9. We can use a look-up table instead of calculating the values, which takes some time. In the AVR, to get the table element we add the index to the address of the look-up table. This is shown in Examples 6-18 through 6-20.

6.4 LOOK-UP TABLE AND TABLE PROCESSING

Example 6-18

Assume that the lower three bits of Port C are connect to three switches. Write a program to send the following ASCII characters to Port D based on the status of the switches.

000	'0'
001	'1'
010	'2'
011	'3'
100	'4'
101	'5'
110	'6'
111	'7'

```
.ORG      $0000
.INCLUDE "M32DEF.INC"
          LDI       R16,0x0
          OUT       DDRC,R16
          LDI       R16,0xFF
          OUT       DDRD,R16
          LDI       ZH,HIGH(ASCII_TABLE<<1)
BEGIN:    IN        R16,PINC
          ANDI      R16,0b00000111
          LDI       ZL,LOW(ASCII_TABLE<<1)
          ADD       ZL,R16
          LPM       R17,Z
          OUT       PORTD,R17
          RJMP      BEGIN
;----------------------------------------
;lookup table for ASCII numbers 0-7
.ORG      0x20
ASCII_TABLE:
          .DB '0','1','2','3','4','5','6','7'
```

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING
6.4 LOOK-UP TABLE AND TABLE PROCESSING

Example 6-19

Write a program to get the x value from Port B and sends x^2 to Port C. Assume that PB3-PB0 has x value of 0-9. use a lookup table instead of a multiply instruction.

```
.ORG     $0000
.INCLUDE "M32DEF.INC"
        LDI     R16,0x0
        OUT     DDRB,R16
        LDI     R16,0xFF
        OUT     DDRC,R16
        LDI     ZH,HIGH(XSQR_TABLE<<1)
L1:     LDI     ZL,LOW(XSQR_TABLE<<1)
        IN      R16,PINB
        ANDI    R16,0x0F
        ADD     ZL,R16
        LPM     R18,Z
        OUT     PORTC,R18
        RJMP    L1
;---------------------------------------
;lookup table for square of numbers 0-9
.ORG     0x10
XSQR_TABLE:
        .DB 0,1,4,9,16,25,36,49,64,81
```

Example 6-20

Write a program to get the x value from Port B and send x^2+2x+3 to Port C. Assume PB3-PB0 has the x value of 0-9. Use a look-up table instead of a multiply instruction.

```
.ORG     $0000
.INCLUDE "M32DEF.INC"
         LDI     R16,0x0
         OUT     DDRB,R16
         LDI     R16,0xFF
         OUT     DDRC,R16
         LDI     ZH,HIGH(TABLE<<1)
L1:      LDI     ZL,LOW(TABLE<<1)
         IN      R16,PINB
         ANDI    R16,0x0F
         ADD     ZL,R16
         LPM     R18,Z
         OUT     PORTC,R18
         RJMP    L1
;---------------------------------------
.ORG     0x10
TABLE:
         .DB 3,11,18,27,38,51,66,83,102
```


mashhoun@iust.ac.ir Iran Univ of Science & Tech

Accessing a look-up table in RAM

The look-up table elements can also be in RAM instead of ROM. Sometimes we need to bring in the elements of the look-up table from RAM because the elements are dynamic and can change. In the AVR, we can do that using the pointers.

Writing table elements in AVR

In AVR we also have the SPM instruction, which allows us to write (store) data into program memory.

6.5 BIT-ADDRESSABILITY

In this section, we provide more programming examples of bit manipulation using the bit-addressable and byte-addressable options of the AVR family.

In Table 6-7, some of the bit-oriented instructions are given. Notice that the bit-oriented instructions use only one addressing mode, the direct addressing mode.

Table 6-7: Single-Bit (Bit-Oriented) Instructions for AVR

Instruction	Function
SBI A,b	Set Bit b in I/O register
CBI A,b	Clear Bit b in I/O register
SBIC A,b	Skip next instruction if Bit b in I/O register is Cleared
SBIS A,b	Skip next instruction if Bit b in I/O register is Set
BST Rr,b	Bit store from register Rr to T
BLD Rd,b	Bit load from T to Rd
SBRC Rr,b	Skip next instruction if Bit b in Register is Cleared
SBRS Rr,b	Skip next instruction if Bit b in Register is Set
BRBS s,k	Branch if Bit s in status register is Set
BRBC s,k	Branch if Bit s in status register is Cleared

Note: A can be any location of the I/O register.

6.5 BIT-ADDRESSABILITY

Manipulating the bits of general purpose registers

Setting the bits

The SBR (Set Bits in Register) instruction sets the specified bits in the general purpose register. It has the following format:

Operation:

(i) $Rd \leftarrow Rd \vee K$

	Syntax:	Operands:	Program Counter:
(i)	SBR Rd,K	$16 \leq d \leq 31,\ 0 \leq K \leq 255$	$PC \leftarrow PC + 1$

16-bit Opcode:

0110	KKKK	dddd	KKKK

Status Register (SREG) and Boolean Formula:

I	T	H	S	V	N	Z	C
–	–	–	⇔	0	⇔	⇔	–

6.5 BIT-ADDRESSABILITY

For example, in the following program the SBR instruction sets the bits 2, 5, and 6 regardless of their previous values.

```
LDI R17,0b01011001    ;R17 = 0x59
SBR R17,0b01100100    ;set bits 2, 5, and 6 in register R17
```

When execution of the above instructions is finished, R17 contains 0x7D. Notice that the SBR instruction is a byte-oriented instruction as it manipulates the whole byte at one time.

6.5 BIT-ADDRESSABILITY

Clearing the bits

The CBR (Clear Bits in Register) instruction clears the specified bits in the general purpose register. It has the following format:

Operation:

(i) $Rd \leftarrow Rd \bullet (\$FF - K)$

	Syntax:	Operands:	Program Counter:
(i)	CBR Rd,K	$16 \leq d \leq 31,\ 0 \leq K \leq 255$	$PC \leftarrow PC + 1$

16-bit Opcode: (see ANDI with K complemented)

Status Register (SREG) and Boolean Formula:

I	T	H	S	V	N	Z	C
–	–	–	⇔	0	⇔	⇔	–

mashhoun@iust.ac.ir Iran Univ of Science & Tech

6.5 BIT-ADDRESSABILITY

For example, in the following program the CBR instruction clears the bits 2, 5, and 6 regardless of their previous values.

```
LDI R17,0b01011001   ;R17 = 0x59
CBR R17,0b01100100   ;clear bits 2, 5, and 6 in register R17
```

After the execution of the above instructions, R17 contains 0x19.

6.5 BIT-ADDRESSABILITY

Copying a bit

As we saw in Chapter 2, one of the bits in the SREG (status register) is named T (temporary), which is used when we want to copy a bit of data from one GPR to another GPR.

BST – Bit Store from Bit in Register to T Flag in SREG

Description:

Stores bit b from Rd to the T Flag in SREG (Status Register).

	Operation:
(i)	$T \leftarrow Rd(b)$

	Syntax:	**Operands:**	**Program Counter:**
(i)	BST Rd,b	$0 \le d \le 31,\ 0 \le b \le 7$	$PC \leftarrow PC + 1$

16-bit Opcode:

1111	101d	dddd	0bbb

mashhoun@iust.ac.ir Iran Univ of Science & Tech

6.5 BIT-ADDRESSABILITY

BLD – Bit Load from the T Flag in SREG to a Bit in Register

Description:

Copies the T Flag in the SREG (Status Register) to bit b in register Rd.

Operation:

(i) $Rd(b) \leftarrow T$

	Syntax:	Operands:	Program Counter:
(i)	BLD Rd,b	$0 \le d \le 31, 0 \le b \le 7$	$PC \leftarrow PC + 1$

16 bit Opcode:

1111	100d	dddd	0bbb

For example, the following program copies bit 3 from R17 to bit 5 in register R19:

```
BST R17,3   ;store bit 3 from R17 to the T flag
BLD R19,5   ;copy the T flag to bit 5 in R19
```

6.5 BIT-ADDRESSABILITY

Example 6-21

A switch is connected to pin PB4. Write a program to get the status of the switch and save it in D0 of internal RAM location 0x200.

Solution:

```
.EQU MYREG = 0x200      ;set aside loc 0x200
      CBI    DDRB,0     ;make PB0 an input
      IN     R17,PINB   ;R17 = PINB
      BST    R17,4      ;T = PINB.4
      LDI    R16,0x00   ;R16 = 0
      BLD    R16,0      ;R16.0 = T
      STS    MYREG,R16  ;copy R16 to location $200
HERE: JMP    HERE
```

6.5 BIT-ADDRESSABILITY

SBRS – Skip if Bit in Register is Set

Description:

This instruction tests a single bit in a register and skips the next instruction if the bit is set.

Operation:

(i) If Rr(b) = 1 then PC ← PC + 2 (or 3) else PC ← PC + 1

	Syntax:	Operands:	Program Counter:
(i)	SBRS Rr,b	$0 \le r \le 31$, $0 \le b \le 7$	PC ← PC + 1, Condition false - no skip PC ← PC + 2, Skip a one word instruction PC ← PC + 3, Skip a two word instruction

16-bit Opcode:

1111	111r	rrrr	0bbb

For example, in the following program the "LDI R20,0x55" instruction will not be executed since bit 3 of R17 is set.

```
LDI  R17,0b00001010
SBRS R17,3   ;skip next instruction if Bit 3 in R17 is set
LDI  R20,0x55
LDI  R30,0x33
```

6.5 BIT-ADDRESSABILITY

SBRC – Skip if Bit in Register is Cleared

Description:

This instruction tests a single bit in a register and skips the next instruction if the bit is cleared.

Operation:

(i) If Rr(b) = 0 then PC $\leftarrow$ PC + 2 (or 3) else PC $\leftarrow$ PC + 1

	Syntax:	Operands:	Program Counter:
(i)	SBRC Rr,b	$0 \le r \le 31$, $0 \le b \le 7$	PC $\leftarrow$ PC + 1, Condition false - no skip PC $\leftarrow$ PC + 2, Skip a one word instruction PC $\leftarrow$ PC + 3, Skip a two word instruction

16-bit Opcode:

1111	110r	rrrr	0bbb

For example, in the following program the "LDI R20, 0x55" instruction will not be executed since bit 2 of R16 is cleared.

```
LDI  R16,0b0001010
SBRC R16,2 ;skip next instruction if Bit 2 in R16 is cleared
LDI  R20,0x55
LDI  R30,0x33
```

6.5 BIT-ADDRESSABILITY

Example 6-22

A switch is connected to pin PC7. Using the SBRS instruction, write a program to check the status of the switch and perform the following:

```
.INCLUDE "M32DEF.INC"   ;include a file according to the IC you use
      CBI    DDRC,7      ;make PC7 an input
      LDI    R16,0xFF
      OUT    DDRD,R16    ;make Port D an output port
AGAIN:IN     R20,PINC    ;R20 = PINC
      SBRS   R20,7       ;skip next line if Bit PC7 is set
      RJMP   OVER        ;it must be LOW
      LDI    R16,'Y'     ;R16 = 'Y' ASCII letter Y
      OUT    PORTD,R16   ;issue R16 to PD
      RJMP   AGAIN       ;we could use JMP instead
OVER: LDI    R16,'N'     ;R16 = 'N' ASCII letter N
      OUT    PORTD,R16   ;issue R16 to PORTD
      RJMP   AGAIN       ;we can use JMP too
```

6.5 BIT-ADDRESSABILITY

Manipulating the bits of I/O registers

As discussed before, we can set and clear the lower 32 I/O registers (addresses 0 to 31) using the SBI (Set bit in 1/O register) and CBI (Clear bit in I/O register) instructions. For example, the following two instructions set the PORTA. 1 and clear the PORTB.4, respectively:

```
SBI  PORTA,1        ;set Bit 1 in PORTA
CBI  PORTB,4        ;clear Bit 4 in PORTB
```

6.5 BIT-ADDRESSABILITY

Example 6-23

Write a program to toggle PB2 a total of 200 times.

Solution:

```
      LDI   R16,200      ;load the count into R16
      SBI   DDRB,2       ;DDRB.1 = 1, make RB1 an output
AGAIN:SBI   PORTB,2      ;set bit PB2 (toggle PB2)
      CBI   PORTB,2      ;clear bit PB2 (toggle PB2)
      DEC   R16          ;decrement R16
      BRNE  AGAIN        ;continue until counter is zero
```

6.5 BIT-ADDRESSABILITY

Status register bit-addressability

Checking a flag bit

BRBS – Branch if Bit in SREG is Set

Description:

Conditional relative branch. Tests a single bit in SREG and branches relatively to PC if the bit is set. This instruction branches relatively to PC in either direction (PC - 63 ≤ destination ≤ PC + 64). The parameter k is the offset from PC and is represented in two's complement form.

Operation:

(i) If SREG(s) = 1 then PC ← PC + k + 1, else PC ← PC + 1

	Syntax:	Operands:	Program Counter:
(i)	BRBS s,k	$0 \le s \le 7$, $-64 \le k \le +63$	PC ← PC + k + 1 PC ← PC + 1, if condition is false

16-bit Opcode:

1111	00kk	kkkk	ksss

Status Register (SREG) and Boolean Formula:

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

6.5 BIT-ADDRESSABILITY

BRBC – Branch if Bit in SREG is Cleared

Description:

Conditional relative branch. Tests a single bit in SREG and branches relatively to PC if the bit is cleared. This instruction branches relatively to PC in either direction (PC - 63 ≤ destination ≤ PC + 64). The parameter k is the offset from PC and is represented in two's complement form.

Operation:

(i) If SREG(s) = 0 then PC ← PC + k + 1, else PC ← PC + 1

	Syntax:	Operands:	Program Counter:
(i)	BRBC s,k	0 ≤ s ≤ 7, -64 ≤ k ≤ +63	PC ← PC + k + 1 PC ← PC + 1, if condition is false

16-bit Opcode:

1111	01kk	kkkk	ksss

Status Register (SREG) and Boolean Formula:

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING

6.5 BIT-ADDRESSABILITY

For example, in the following program the LDI instruction is not executed when the carry flag is set:

```
        BRBS  0,L1        ;branch if status flag bit 0 is set
        LDI   R20,3
  L1:
```

We can write the same program using the "BRCS LI" instruction as follows:

```
        BRCS  L1          ;branch if carry flag is set
        LDI   R20,3
  L1:
```

6.5 BIT-ADDRESSABILITY

Manipulating a bit

BSET – Bit Set in SREG

Description:

Sets a single Flag or bit in SREG.

Operation:

(i) SREG(s) ← 1

	Syntax:	Operands:	Program Counter:
(i)	BSET s	$0 \le s \le 7$	PC ← PC + 1

16-bit Opcode:

1001	0100	0sss	1000

Status Register (SREG) and Boolean Formula:

I	T	H	S	V	N	Z	C
⇔	⇔	⇔	⇔	⇔	⇔	⇔	⇔

6.5 BIT-ADDRESSABILITY

BCLR – Bit Clear in SREG

Description:

Clears a single Flag in SREG.

Operation:

(i) SREG(s) $\leftarrow 0$

	Syntax:	**Operands:**	**Program Counter**
(i)	BCLR s	$0 \leq s \leq 7$	$PC \leftarrow PC + 1$

16-bit Opcode:

1001	0100	1sss	1000

Status Register (SREG) and Boolean Formula:

I	T	H	S	V	N	Z	C
⇔	⇔	⇔	⇔	⇔	⇔	⇔	⇔

6.5 BIT-ADDRESSABILITY

For example, the following instruction sets the carry flag.

```
BSET  0            ;set bit 0 (carry flag)
```

As another example, the instruction "BSET 2" sets the N (Negative) flag.

For example, the following instruction clears the carry flag.

```
BCLR  0            ;clear bit 0 (carry flag)
```

As another example, the instruction "BCLR 1" clears the Z (Zero) flag.

6.5 BIT-ADDRESSABILITY

A more convenient way is to use the CLZ instruction, as shown in Table 6-9.

Table 6-9: Manipulating the Flags of the Status Register

Instruction	Action		Instruction	Action	
SEC	Set Carry	C = 1	CLC	Clear Carry	C = 0
SEZ	Set Zero	Z = 1	CLZ	Clear Zero	Z = 0
SEN	Set Negative	N = 1	CLN	Clear Negative	N = 0
SEV	Set overflow	V = 1	CLV	Clear overflow	V = 0
SES	Set Sign	S = 1	CLS	Clear Sign	S = 0
SEH	Set Half carry	H = 1	CLH	Clear Half carry	H = 0
SET	Set Temporary	T = 1	CLT	Clear Temporary	T = 0
SEI	Set Interrupt	I = 1	CLI	Clear Interrupt	I = 0

6.5 BIT-ADDRESSABILITY

Internal RAM bit-addressability

The internal RAM is not bit-addressable.

So, in order to manipulate a bit of the internal RAM location, you should bring it into the general purpose register and then manipulate it, as shown in Examples .

Example 6-25

Write a program to see if the internal RAM location $195 contains an even value. If so, send it to Port B. If not, make it even and then send it to Port B.

6.5 BIT-ADDRESSABILITY

Solution 1:

```
.EQU  MYREG = 0x195           ;set aside loc 0x195
      LDI   R16,0xFF
      OUT   DDRB,R16          ;make Port B an output port
AGAIN:LDS   R16,MYREG
      SBRS  R16,0             ;bit test D0, skip if set
      RJMP  OVER              ;it must be LOW
      CBR   R16,0b00000001    ;clear bit D0 = 0
OVER: OUT   PORTB,R16         ;copy it to Port B
      JMP   AGAIN             ;we can use RJMP too
```

Solution 2:

```
.EQU  MYREG = 0x195           ;set aside loc 0x195
      LDI   R16,0xFF
      OUT   DDRB,R16          ;make Port B an output port
AGAIN:LDS   R16,MYREG
      CBR   R16,0b00000001    ;clear bit D0 = 0
OVER: OUT   PORTB,R16         ;copy it to Port B
      JMP   AGAIN             ;we can use RJMP too
```

6.5 BIT-ADDRESSABILITY

Example 6-27

Write a program to see if the internal RAM location $137 contains an even value. If so, write 0x55 into location $200. If not, write 0x63 into location $200.

Solution:

```
.EQU  MYREG = 0x137       ;set aside location 0x137
.EQU  RESULT= 0x200
      LDS   R16,MYREG
      SBRC  R16,0         ;skip if clear Bit D0 of R16 register is clr
      RJMP  OVER          ;it is odd
      LDI   R16,0x55
      STS   RESULT,R16
      RJMP  HERE
OVER: LDI   R16,0x63
      STS   RESULT,R16
HERE: RJMP  HERE
```

6.6 ACCESSING EEPROM IN AVR

6.6 ACCESSING EEPROM IN AVR

The data in SRAM will be lost if the power is disconnected. However, we need a place to save our data to protect them against power failure. EEPROM memory can save stored data even when the power is cut off. In this section we will show how to write to EEPROM memory and how to access it.

Table 6-10: Size of EEPROM Memory in ATmega Family

Chip	Bytes	Chip	Bytes	Chip	Bytes
ATmega8	512	ATmega16	512	ATmega32	1024
ATmega64	2048	ATmega128	4096	ATmega256RZ	4096
ATmega640	4096	ATmega1280	4096	ATmega2560	4096

6.6 ACCESSING EEPROM IN AVR

EEPROM registers

There are three I/O registers that are directly related to EEPROM. These are **EECR** (EEPROM Control Register), **EEDR** (EEPROM Data Register), and **EEARH-EEARL** (EEPROM Address Register High-Low). Each of these registers is discussed in detail in this section.

EEPROM Data Register (EEDR)

To write data to EEPROM, you have to write it to the EEDR register and then transfer it to EEPROM. Also, if you want to read from EEPROM you have to read from EEDR. In other words, EEDR is a bridge between EEPROM and CPU.

6.6 ACCESSING EEPROM IN AVR

EEPROM Address Register (EEARH and EEARL)

The EEARH:EEARL registers together make a 16-bit register to address each location in EEPROM memory space. When you want to read from or write to EEPROM, you should load the EEPROM location address in EEARs. As you see in Figure 6-15, only 10 bits of the EEAR registers are used in ATmega32. Because ATmega32 has 1024-byte EEPROM locations, we need 10 bits to address each location in EEPROM space

Bit	15	14	13	12	11	10	9	8
EEARH	-	-	-	-	-	-	EEAR9	EEAR8
EEARL	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0
Bit	7	6	5	4	3	2	1	0

Figure 6-15. EEPROM Address Registers

6.6 ACCESSING EEPROM IN AVR

EEPROM Control Register (EECR)

The EECR register is used to select the kind of operation to perform on. The operation can be start, read, and write. In Figure 6-16 you see the bits of the EECR register. The bits are as follows:

EECR	-	-	-	-	EERIE	EEMWE	EEWE	EERE
Bit	7	6	5	4	3	2	1	0

Figure 6-16. EEPROM Control Registers

EEPROM Read Enable (EERE): Setting this bit to one will cause a read operation if EEWE is zero. When a read operation starts, one byte of EEPROM will be read into the EEPROM Data Register (EEDR). The EEAR register specifies the address of the desired byte.

6.6 ACCESSING EEPROM IN AVR

EEPROM Write Enable (EEWE) and EEPROM Master Write Enable (EEMWE): When EEMWE is set, setting EEWE within four clock cycles will start a write operation. If EEMWE is zero, setting EEWE to one will have no effect.

When you set EEMWE to one, the hardware clears the bit to zero after four clock cycles. This prevents unwanted write operations on EEPROM contents. Notice that you cannot start read or write operations before the last write operation is finished. You can check for this by polling the EEWE bit. If EEWE is zero it means that EEPROM is ready to start a new read or write operation.

EEPROM Ready Interrupt Enable (EERIE): In Chapter 10 you will learn about interrupts in AVR.

As you see in Figure 6- 16, bits 4 to 7 of EECR are unused at the present time and are reserved.

6.6 ACCESSING EEPROM IN AVR

Programming the AVR to write on EEPROM

To write on EEPROM the following steps should be followed. Notice that steps 2 and 3 are optional, and the order of the steps is not important. Also note that you cannot do anything between step 4 and step 5 because the hardware clears the EEMWE bit to zero after four clock cycles.

1. Wait until EEWE becomes zero.
2. Write new EEPROM address to EEAR (optional).
3. Write new EEPROM data to EEDR (optional).
4. Set the EEMWE bit to one (in EECR register).
5. Within four clock cycles after setting EEMWE, set EEWE to one.

See Example 6-28 to see how we write a byte on EEPROM.

6.6 ACCESSING EEPROM IN AVR

Example 6-28

Write an AVR program to store 'G' into location 0x005F of EEPROM .

Solution:

```
.INCLUDE "M16DEF.INC"

WAIT:               ;wait for last write to finish
SBIC  EECR,EEWE     ;check EEWE to see if last write is finished
RJMP  WAIT          ;wait more
LDI   R18,0         ;load high byte of address to R18
LDI   R17,0x5F      ;load low byte of address to R17
OUT   EEARH, R18    ;load high byte of address to EEARH
OUT   EEARL, R17    ;load low byte of address to EEARL
LDI   R16,'G'       ;load 'G' to R16
OUT   EEDR,R16      ;load R16 to EEPROM Data Register
SBI   EECR,EEMWE    ;set Master Write Enable to one
SBI   EECR,EEWE     ;set Write Enable to one
```

Run and simulate the code on AVR Studio to see how the content of the EEPROM changes after the last line of code. Enter four NOP instructions before the last line, change the 'G' to 'H', and run the code again. Explain why the code doesn't store 'H' at location 0x005F of EEPROM.

6.6 ACCESSING EEPROM IN AVR

Programming the AVR to read from EEPROM

To read from EEPROM the following steps should be taken. Note that step 2 is optional.

1. Wait until EEWE becomes zero.
2. Write new EEPROM address to EEAR (optional).
3. Set the EERE bit to one.
4. Read EEPROM data from EEDR.

See Example 6-29 to see how we read a byte from EEPROM.

6.6 ACCESSING EEPROM IN AVR

Example 6-29

Write an AVR program to read the content of location 0x005F of EEPROM into PORTB.

Solution:

```
.INCLUDE "M16DEF.INC"
        LDI     R16,0xFF
        OUT     DDRB,R16
WAIT:                           ;wait for last write to finish
        SBIC    EECR,EEWE       ;check EEWE to see if last write is finished
        RJMP    WAIT            ;wait more
        LDI     R18,0           ;load high byte of address to R18
        LDI     R17,0x5F        ;load low byte of address to R17
        OUT     EEARH, R18      ;load high byte of address to EEARH
        OUT     EEARL, R17      ;load low byte of address to EEARL
        SBI     EECR,EERE       ;set Read Enable to one
        IN      R16,EEDR        ;load EEPROM Data Register to R16
        OUT     PORTB,R16       ;out R16 to PORTB
```

6.6 ACCESSING EEPROM IN AVR

Initializing EEPROM

In Section 6-4, you saw how to allocate program memory using the .DB directive. We can also allocate and initialize the EEPROM using the .DB directive. If we write .ESEG before a definition, the variable will be located in the EEPROM, whereas .CSEG before a definition causes the variable to be allocated in the code (program) memory. By default the variables are located in the program memory. For example, the following code allocates locations $10 and $11 of EEPROM for DATAl and DATA2, and initializes them with $95 and $19, respectively:

```
        .ESEG
        .ORG $10
DATA1:        .DB   $95
DATA2:        .DB   $19
```

6.6 ACCESSING EEPROM IN AVR

The following code allocates DATA1 and DATA3 in program memory and DATA2 in EEPROM:

```
DATA1:       .DB    $10    ;by default it is located in code memory
       .ESEG
DATA2:       .DB    $20    ;it is located in EEPROM
DATA3:       .DB    $35    ;it is located in EEPROM
       .CSEG
DATA4:       .DB    $45    ;it is located in code memory
```

Example 6-30

Write a program that counts how many times a system has been powered up.

Solution:

```
.INCLUDE "M32DEF.INC"
      LDI    R20,HIGH(RAMEND)
      OUT    SPH,R20
      LDI    R20,LOW(RAMEND)
      OUT    SPL,R20             ;initialize stack pointer
      LDI    XH,HIGH(COUNTER)
      LDI    XL,LOW(COUNTER)     ;X points to COUNTER
      CALL   LOAD_FROM_EEPROM    ;load R20 with value of COUNTER
      INC    R20                 ;increment R20
      CALL   STORE_IN_EEPROM     ;store R20 in EEPROM
HERE: RJMP   HERE
```

6.6 ACCESSING EEPROM IN AVR

```
;-----Load R20 with contents of location X of EEPROM
LOAD_FROM_EEPROM:
      SBIC   EECR, EEWE
      RJMP   LOAD_FROM_EEPROM   ;wait while EEPROM is busy
      OUT    EEARH,XH
      OUT    EEARL,XL           ;EEAR = X
      SBI    EECR,EERE          ;set Read Enable to one
      IN     R20,EEDR           ;load EEPROM Data Register to r20
      RET

;-----Store R20 into location X of EEPROM
STORE_IN_EEPROM:
      SBIC   EECR, EEWE
      RJMP   STORE_IN_EEPROM    ;wait while EEPROM is busy
      OUT    EEARH,XH
      OUT    EEARL,XL           ;EEAR = X
      OUT    EEDR,R20
      SBI    EECR,EEMWE         ;set Master Write Enable to one
      SBI    EECR,EEWE          ;write EEDR into EEPROM
      RET
;-------EEPROM
.ESEG
.ORG 0
COUNTER:    .DB    0
```

COUNTER is initialized with \$0. Then, it is incremented on each power-up.

6.7 CHECKSUM AND SUNROUTINES

Checksum byte in EEPROM

The checksum will detect any corruption of the contents of ROM. One cause of ROM corruption is current surge, either when the system is turned on, or during operation. To ensure data integrity in ROM, the checksum process uses what is called a ***checksum byte. The checksum byte is an extra byte*** that is tagged to the end of a series of bytes of data. To calculate the checksum byte of a series of bytes of data, the following steps can be taken:

1. Add the bytes together and drop the carries.
2. Take the 2's complement of the total sum, and that is the checksum byte, which becomes the last byte of the series.

To perform a checksum operation, add all the bytes, including the checksum byte. The result must be zero. If it is not zero, one or more bytes of data have been changed (corrupted).

6.7 CHECKSUM AND SUNROUTINES

Checksum program

The checksum generation and testing program is given in subroutine form. Five subroutines perform the following operations:

1. Retrieve the data from EEPROM.
2. Test the checksum byte for any data error.
3. Initialize variables if the checksum byte is corrupted.
4. Calculate the checksum byte.
5. Store the data in EEPROM.

Each of these subroutines can be used in other applications.

6.7 CHECKSUM AND SUNROUTINES

Example 6-31 shows how to manually calculate the checksum for a list of values.

Example 6-31

Assume that we have 4 bytes of hexadecimal data: \$25, \$62, \$3F, and \$52.

(a) Find the checksum byte.
(b) Perform the checksum operation to ensure data integrity.
(c) If the second byte, \$62, has been changed to \$22, show how the checksum method detects the error.

Solution:

(a) Find the checksum byte.

```
   $25
 + $62
 + $3F
 + $52
  $118
```

(Dropping the carry of 1, we have \$18. Its 2's complement is \$E8. Therefore, the checksum byte is \$E8.)

6.7 CHECKSUM AND SUNROUTINES

(b) Perform the checksum operation to ensure data integrity.

```
    $25
  + $62
  + $3F
  + $52
  + $E8
   $200     (Dropping the carries, we see 00, indicating that the data is not corrupted.)
```

(c) If the second byte, \$62, has been changed to \$22, show how the checksum method detects the error.

```
    $25
  + $22
  + $3F
  + $52
  + $E8
   $1C0     (Dropping the carry, we get $C0, which is not 00. This means that the data
            is corrupted.)
```

6.7 CHECKSUM AND SUNROUTINES

BCD to ASCII conversion program

Many RTCs (real-time clocks) provide time and date in BCD format. To display the BCD data on an LCD or a PC screen, we need to convert it to ASCII. Program 6-2

(a) transfers packed BCD data from program ROM to data RAM,

(b) converts packed BCD to ASCII, and

(c) sends the ASCII to port B for display.

6.7 CHECKSUM AND SUNROUTINES

Binary (hex) to ASCII conversion program

Many ADC (analog-to-digital converter) chips provide output data in binary (hex). To display the data on an LCD or PC screen, we need to convert it to ASCII. The code for the binary-to-ASCII conversion is shown in Program **6-3.**

Notice that the subroutine gets a byte of 8-bit binary (hex) data from Port B and converts it to decimal digits, and the second subroutine converts the decimal digits to ASCII digits and saves them.

We are saving the low digit in the lower address location and the high digit in higher address location.

This is referred to as the little-endian convention (i.e., low byte to low location, and high byte to high location).

6.8 MACROS

What is a macro and how is it used?

Macros allow the programmer to write the task (code to perform a specific job) once only, and to invoke it whenever it is needed.

Macro definition

Every macro definition must have three parts, as follows:

```
.MACRO    name
        .........
.ENDMACRO
```

What goes between the .MACRO and .ENDMACRO directives is called the ***body*** of the macro. The name must be unique and must follow Assembly language naming conventions. A macro can take up to 10 parameters. The parameters can be referred to as *@0* to *@9* in the body of the macro.

6.8 MACROS

For example, moving immediate data into I/O register data RAM is a widely used service, but there is no instruction for that. We can use a macro to do the job as shown in the following code:

```
.MACRO      LOADIO
      LDI   R20,@1
      OUT   @0,R20
.ENDMACRO
```

The following are three examples of how to use the above macro:

```
1. LOADIO   PORTA, 0x20       ;send value 0x20 to PORTA

2. .EQU     VAL_1 = 0xFF
   LOADIO   DDRC, VAL_1
3. LOADIO   SPL, 0x55         ;send value $55 to SPL
```

AMMING

6.8 M

Now exa

```
;Program 6-4: toggling Port B using macros
;-----------------------------------------
.INCLUDE "M32DEF.INC"
.MACRO  LOADIO
        LDI    R20,@1
        OUT    @0,R20
.ENDMACRO

;--------------------------time delay macro
.MACRO  DELAY
        LDI @0,@1
BACK:
        NOP
        NOP
        NOP
        NOP
        DEC    @0
        BRNE  BACK
.ENDMACRO

;--------------------------program starts
        .ORG   0
        LOADIO   DDRB,0xFF        ;make PORTB output
L1:     LOADIO   PORTB,0x55       ;PORTB = 0x55
        DELAY    R18,0x70         ;delay
        LOADIO   PORTB,0xAA       ;PB = 0xAA
        DELAY    R18,0x70         ;delay
        RJMP     L1
```

6.8 MACROS

.INCLUDE directive

The .INCLUDE directive allows a programmer to write macros and save them in a file, and later bring them into any program file. For example, assume that the following widely used macros were written and then saved under the filename **"MYMACRO1.MAC"**.

Assuming that the LOADIO and DELAY macros are saved on a disk under the filename "MYMACRO.MAC", the **.INCLUDE** directive can be used to bring this file into any ".asm" file and then the program can call upon any of the macros as many times as needed. When a file includes all macros, the macros are listed at the beginning of the ".lst" file and, as they are expanded, will be part of the program.

6.8 MACROS

To understand this, see Program ***6-5.***

```
;Program 6-5: toggling Port B using macros
.INCLUDE "M32DEF.INC"
.INCLUDE "MYMACRO1.MAC" ;get macros from macro file
;------------------------------program starts
      .ORG  0
      LOADIO   DDRB,0xFF
L1:   LOADIO   PORTB,0x55
      DELAY    R18,0x70
      LOADIO   PORTB,0xAA
      DELAY    R18,0x70
      RJMP     L1
```

6.8 MACROS

. LISTMAC directive

Using the .LISTMAC directive we can **turn on the display of the bodies of macros in the list file. For example, examine** the following code:

```
.INCLUDE "M32DEF.INC"
.MACRO      LOADIO
      LDI   R20,@1
      OUT   @0,R20
.ENDMACRO

      LOADIO      PORTA,0x20
      LOADIO      DDRA,0x53
HERE: JMP         HERE
```

The assembler provides the following code in the .lst file:

```
                        .MACRO      LOADIO
                              LDI   R20,@1
                              OUT   @0,R20
                        .ENDMACRO
000000 e240
000001 bb4b             LOADIO      PORTA,0x20
000002 e543
000003 bb4a             LOADIO      DDRA,0x53
000004 940c 0004 HERE:JMP           HERE
```

6.8 MACROS

If we add the .LISTMAC directive to the above code:

```
.MACRO      LOADIO
      LDI   R20,@1
      OUT   @0,R20
.ENDMACRO
.LISTMAC
      LOADIO      PORTA,0x20
      LOADIO      DDRA,0x53
HERE:JMP    HERE
```

```
                          .MACRO     LOADIO
                                LDI  R20,@1
                                OUT  @0,R20
                          .ENDMACRO
                          .LISTMAC
                         +
000000 e240              +LDI R20 , 0x20
000001 bb4b              +OUT PORTA , R20
                                LOADIO      PORTA,0x20
                         +
000002 e543              +LDI R20 , 0x53
000003 bb4a              +OUT DDRA , R20
                                LOADIO      DDRA,0x53
000004 940c 0004 HERE:JMP     HERE
```

The + indicates that the code is from the macro.