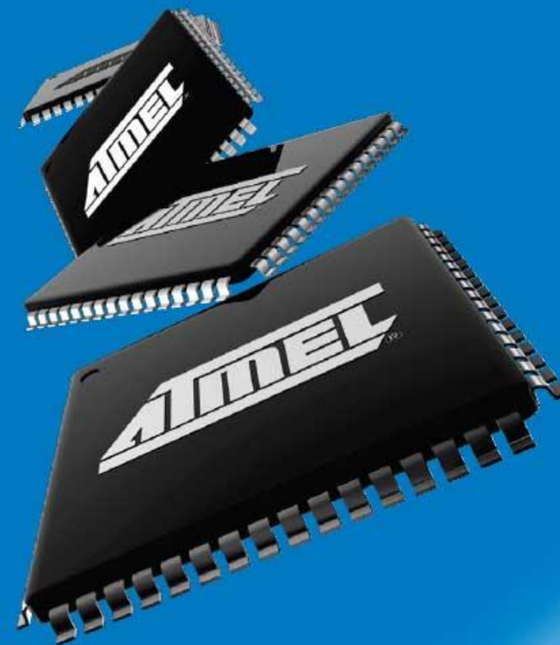
AVR®

8-bit Microcontrollers

AVR®32

32-bit Microcontrollers and Application Processors



Addressing Modes

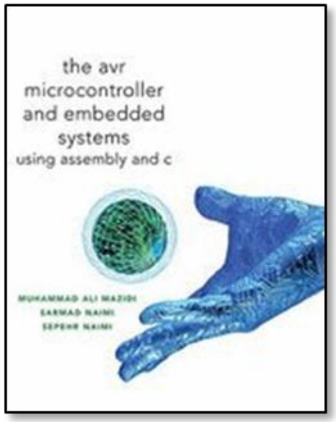
February 2009

Atmel®

Everywhere You Are®

Addressing Modes Part II – AVR Addressing Indirect

READING



The AVR Microcontroller and Embedded Systems using Assembly and C)
by Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi

Chapter 6: AVR Advanced Assembly Language Programming

- Section 6.1: Introducing some more assembler directives
- Section 6.3: Register Indirect Addressing Mode
- Section 6.4: Look-up Table and Table Processing

Contents

ADDRESSING MODES

- When loading and storing data we have several ways to "address" the data.
- The AVR microcontroller supports addressing modes for access to the Program memory (Flash) and Data memory (SRAM, Register file, I/O Memory, and Extended I/O Memory).

Load-store Instructions

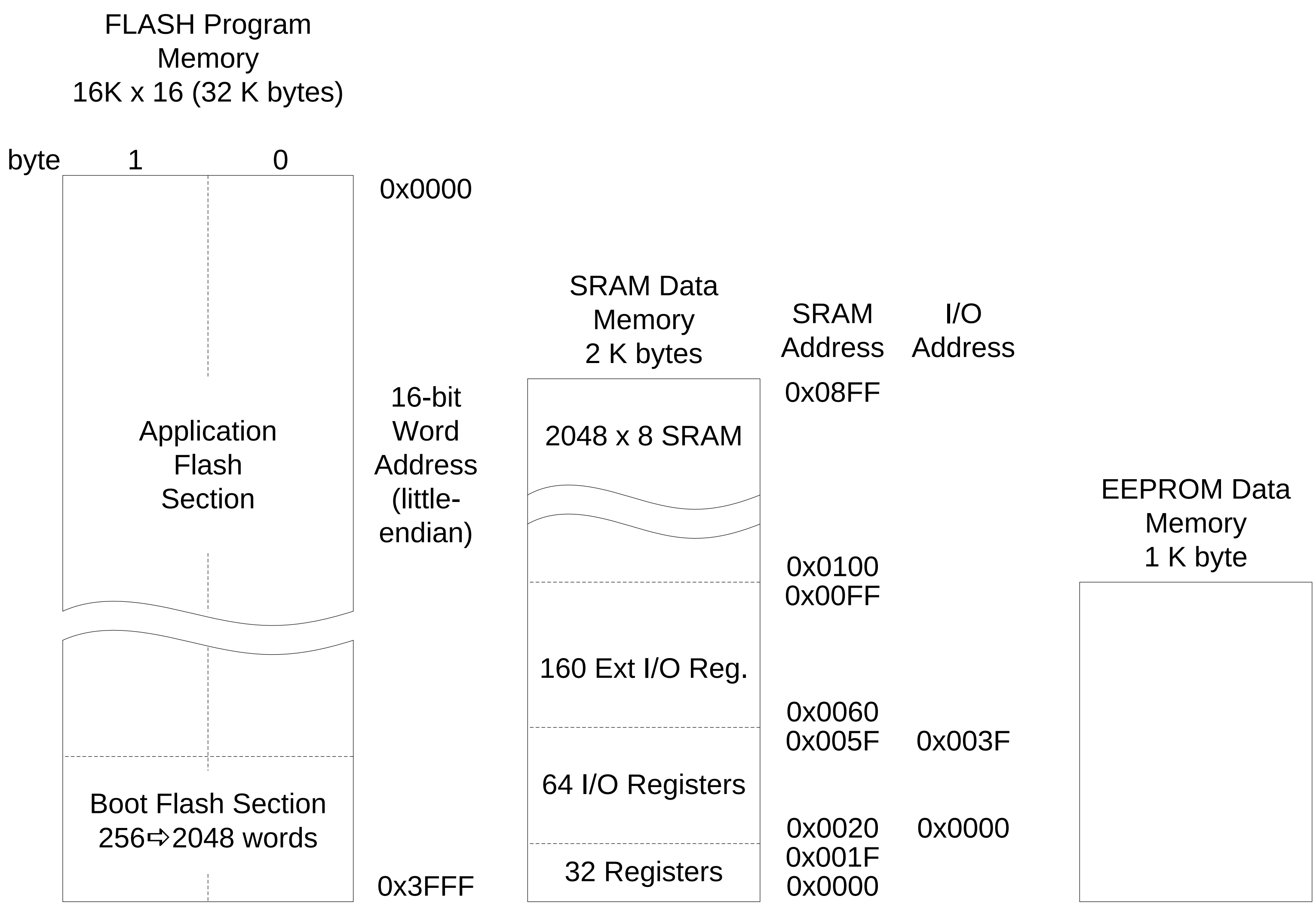
	Address Space		
Addressing Mode	Flash Program	SRAM Data	I/O
Immediate	`ldi`		
Direct		`lds, sts`	`in, out`
Indirect	`lpm, spm (1)`	`ld, st (2)`	
Indirect with Displacement		`ldd, std (3)`	

Notes:

1. Load-store indirect from program memory to register using index register z. Index register can be unchanged, or post-incremented. The program memory is organized in 16-bit words while the Z-pointer is a byte address. Byte ordering is little-endian.
2. Load-store indirect from data space to register using index registers x, y, and z. Index register can be unchanged, pre-decrement, or post-incremented.
3. Load-store indirect with displacement from data space to register using index registers y and z.

Operand Locations and the ATmega328P Memory Model

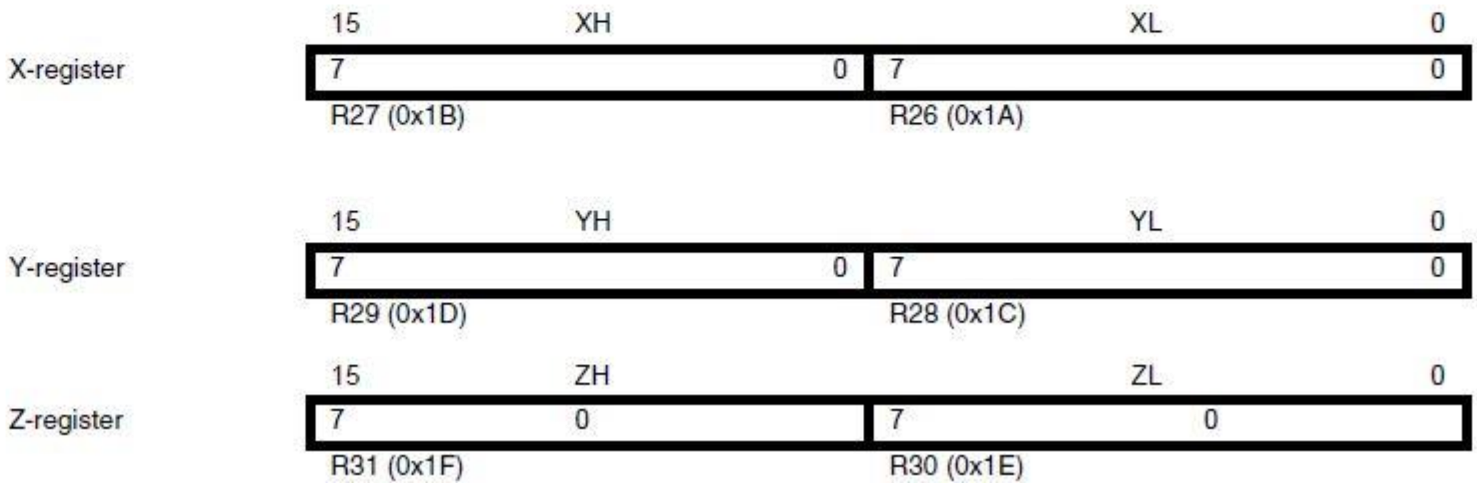
- When selecting an addressing mode you should ask yourself where is the operand (data) located within the memory model of the AVR processor and when do I know its address (assembly time or at run time).



Source: http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf 8-bit AVR Instruction Set

The X-register, Y-register, and Z-register

The registers R26..R31 have some added functions to their general purpose usage. These registers are 16-bit address pointers for indirect addressing of the data space. The three indirect address registers X, Y, and Z are defined as described here.



In the different addressing modes these address registers have functions as fixed displacement, automatic increment, and automatic decrement (see the instruction set reference for details).

PROGRAM MEMORY INDIRECT

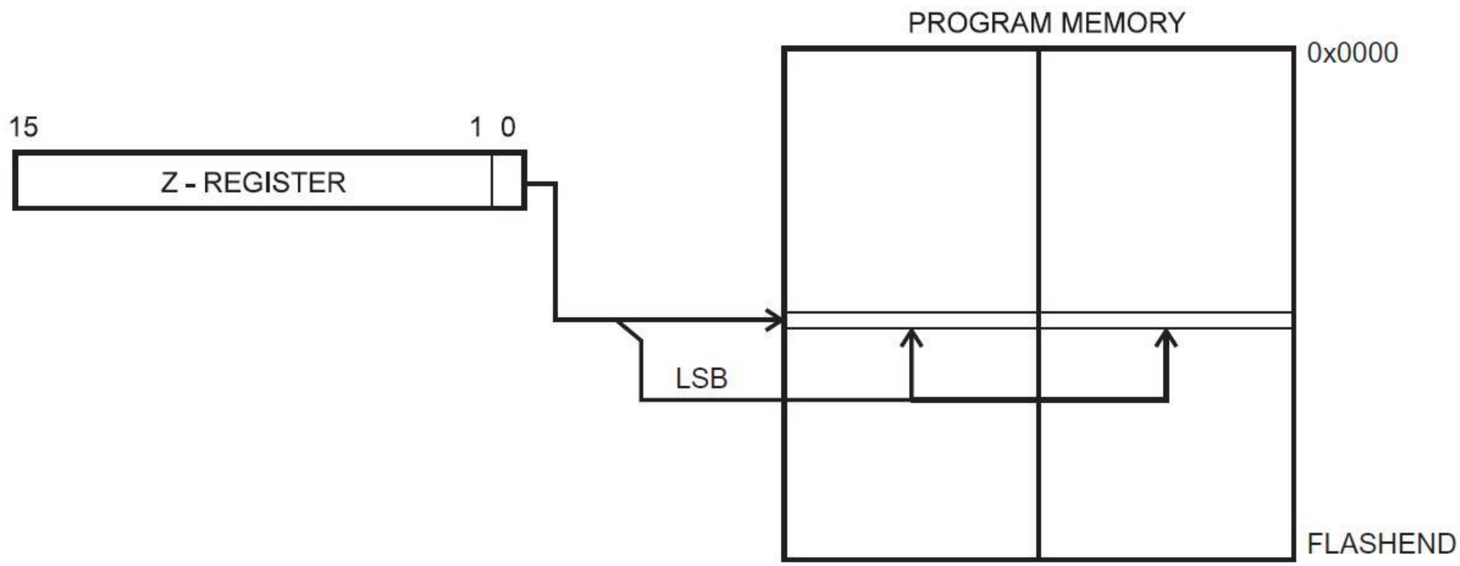
- The indirect addressing mode in all its forms is used when you will not know the location of the data you want until the program is running. For example, in our 7-segment decoder example, we do not know ahead of time which number (0 to F) we want to decode.
- The most significant bit of the ZH:ZL is lost, in order to make space for the byte address in the least significant bit.

```
lpm     Rd, Z
```

Instruction Encoding

15 – 12	11 – 8	7 – 4	3 – 0
1001	000d	dddd	0100

Addressing Mode Operation



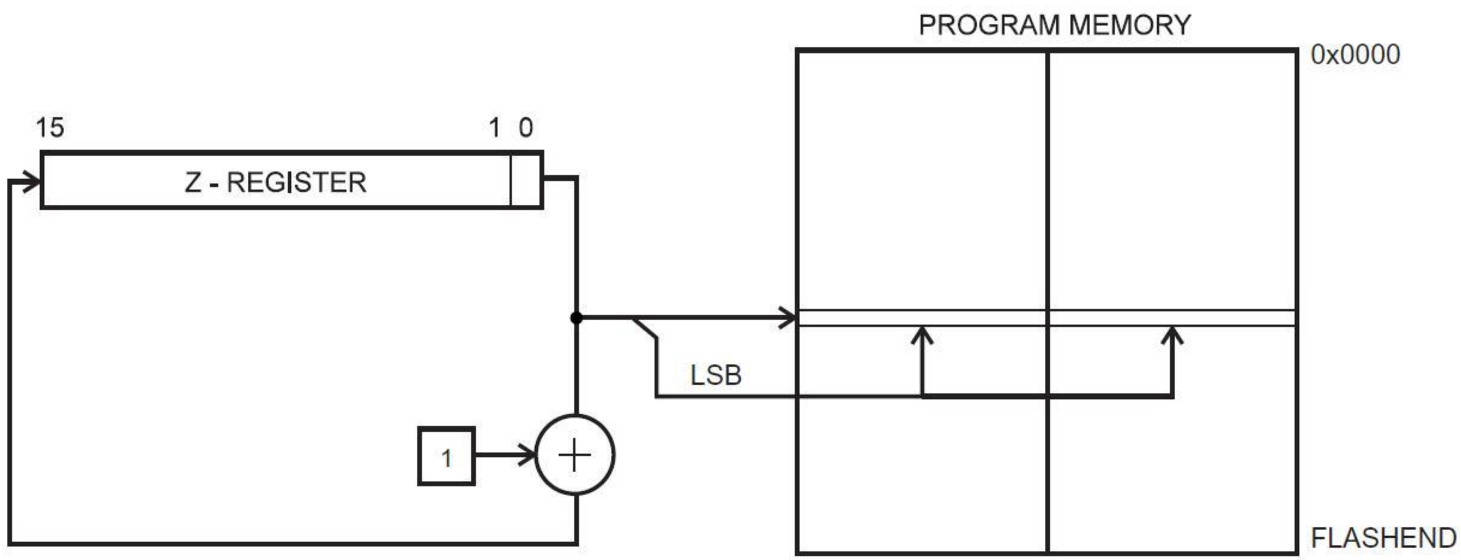
PROGRAM MEMORY INDIRECT WITH POST-INCREMENT

```
lpm     r16, Z+
```

Instruction Encoding

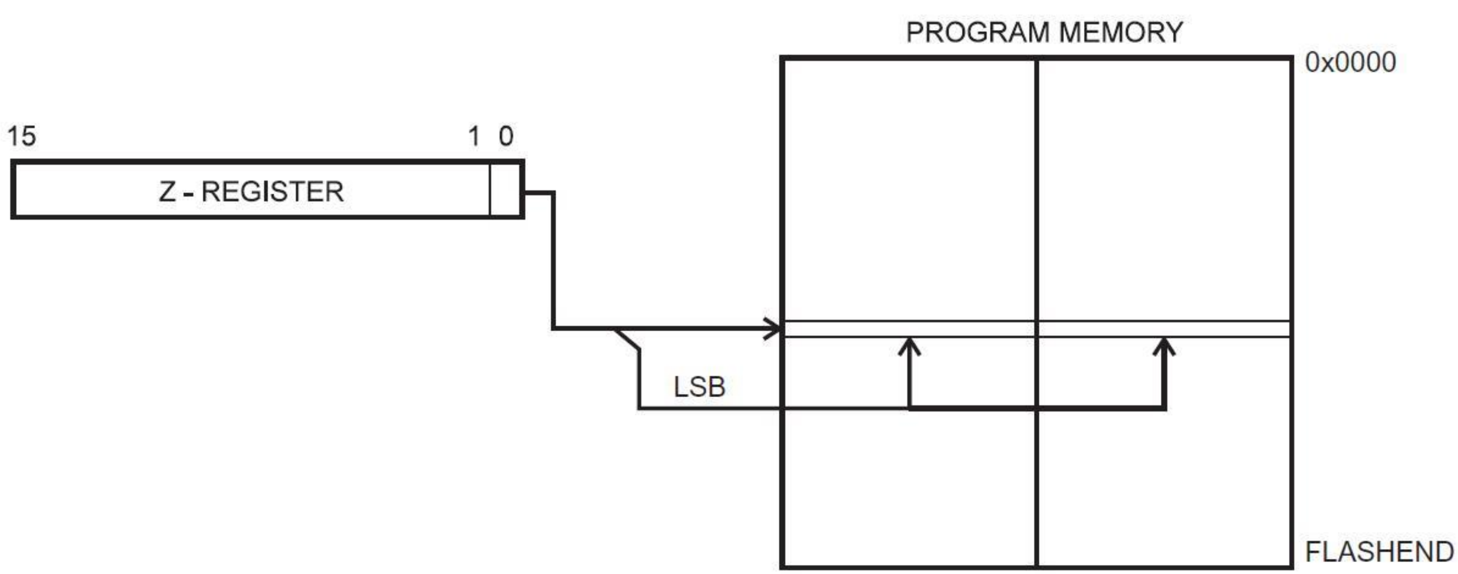
15 12	11 8	7 4	3 0
1001	000d	dddd	0101

Addressing Mode Operation



PROGRAM MEMORY INDIRECT – EXAMPLE 1

```
ldi ZH, high(Table<<1)    // Initialize Z-pointer (read next page)
ldi ZL, low(Table<<1)
lpm r16, Z                // Load constant from Program
; Memory pointed to by Z (r31:r30)
...
Table:
.DW 0x063F    // 0x3F is addressed when Z_LSB = 0
              // 0x06 is addressed when Z_LSB = 1
```



Princeton versus *Modified* Harvard Memory Models

Princeton or Von Neumann Memory Model

Program and data share the same memory space. Processors used in all personal computers, like the Pentium, implement a von Neumann architecture.

Harvard Memory Model

As we have learned in the Harvard Memory Model, program and data memory are separated. The AVR processors among others including the Intel 8051 use this memory model. One advantage of the Harvard architecture for microcontrollers is that program memory can be wider than data memory. This allows the processor to implement more instructions while still working with 8-bit data. For the AVR processor program memory is 16-bits wide while data memory is only 8-bits.

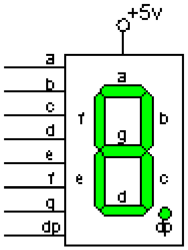
You may have already noticed that when you single step your program in the simulator of AVR Studio the Program Counter is incremented by 1 each time most instructions are executed. No surprise there right? Wrong. The program memory of the AVR processor can also be accessed at the byte level. In most cases this apparent paradox is transparent to the operation of your program with one important exception. That important exception is occurs when you want to access data stored in program memory. It is this ability of the AVR processor to access data stored in program memory that makes it a "Modified" Harvard Memory Model.

When you access from program memory you will be working with byte addresses not words (16-bits). The assembler is not smart enough to know the difference and so when you ask for an address in program memory it returns its word address. To convert this word address into a byte address you need to multiply it by 2. Problematically we do this by using the shift left syntax of C++ to explicitly tell the assembler to multiply the word address by 2. Remember, when you shift left one place you are effectively multiplying by 2.

With this in mind, we would interpret the following AVR instruction as telling the AVR assembler to convert the word address of label beehives in program memory to a byte address and then to take the low order of the resulting value and put into the source operand of the instruction.

```
ldi    ZL,low(beeHives<<1)    // load word address of beeHives look-up
```

PROGRAM MEMORY INDIRECT – EXAMPLE 2



- Program Memory Indirect is great for implementing look-up tables located in Flash program memory – including decoders (gray code → binary, hex → seven segment, …)
- In this example I build a 7-segment decoder in software.

```
BCD_to_7SEG:
    ldi     r16, 0b00001111      // limit to least significant
    and     r0, r16              // nibble (4 bits)
    ldi     ZL,low(table<<1)     // load address of look-up
    ldi     ZH,high(table<<1)
    clr     r1
    add     ZL, r0
    adc     ZH, r1
    lpm     spi7SEG, Z
    ret
//                   gfedcba     gfedcba     gfedcba
    table: DB   0b00111111, 0b00000110, 0b01011011, …
//               0           1           2
```

BIG ENDIAN VERSUS LITTLE ENDIAN – DEFINE BYTE

- To help understand the difference between Big and Little Endian let's take a closer look at how data is stored in Flash Program Memory. We will first look at the Define Byte (.DB) Assembly Directive and then at the Define Word (.DW) Assembly Directive.

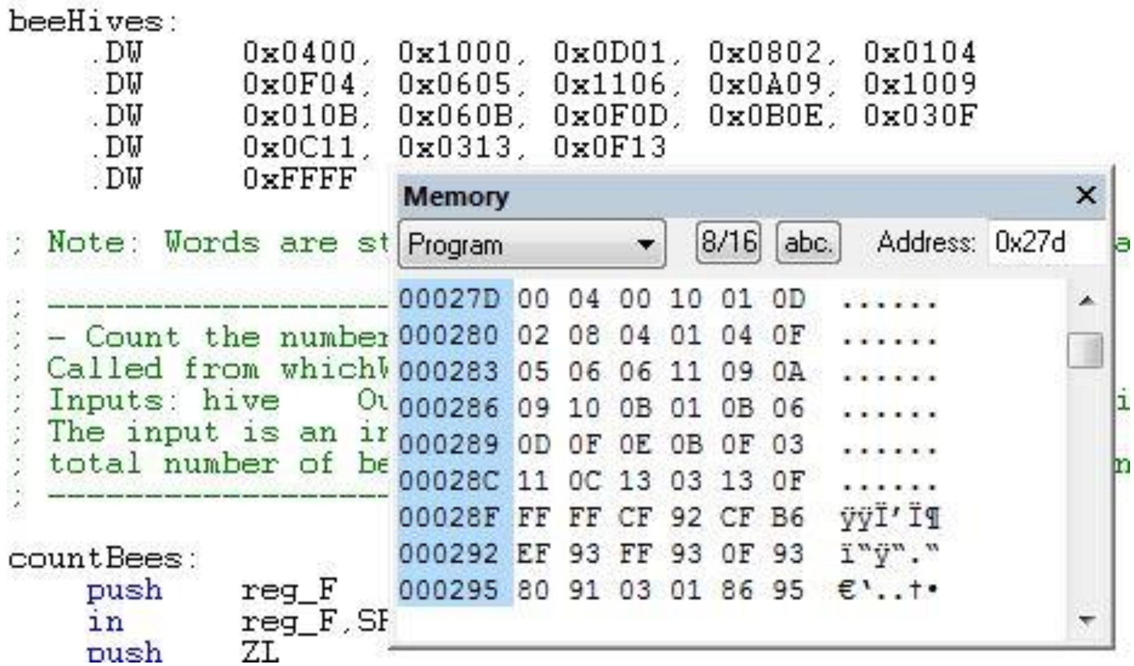
```
                   //            gfedcba     gfedcba     gfedcba     gfedcba     gfedcba     gfedcba
000036 063f
000037 4f5b
000038 6d66        table:  .DB   0b00111111, 0b00000110, 0b01011011, 0b01001111, 0b01100110, 0b01101101
                   //            0           1           2           3           4           5
000039 077d
00003a 677f
00003b 7c77                .DB   0b01111101, 0b00000111, 0b01111111, 0b01100111, 0b01110111, 0b01111100
                   //            6           7           8           9           A           B
00003c 5e39
00003d 7179                .DB   0b00111001, 0b01011110, 0b01111001, 0b01110001
                   //            C           D           E           F
```

Each table entry (.DB) contains one byte. If we look at the first table entry we see 0b00111111 which corresponds to 3f in hexadecimal. Comparing this with the corresponding address and data fields on the left... Wait a minute - where did 06 come from? That the second entry in the table (0b00000110 = 06_{16}). The bytes are backwards and here is why.

There are two basic ways information can be saved in memory known as Big Endian and Little endian. For Big Endian the most significant byte (big end) is saved in the lowest order byte; so 0x3f06 would be saved as bytes 0x3f and 0x06. For Little Endian the least significant byte (little end) is saved in the lowest order byte; so 0x3f06 is save as bytes 0x06 and 0x3f. As you hopefully have guessed by now the AVR processor is designed to work with data words saved as little endian.

BIG ENDIAN VERSUS LITTLE ENDIAN – DEFINE WORD

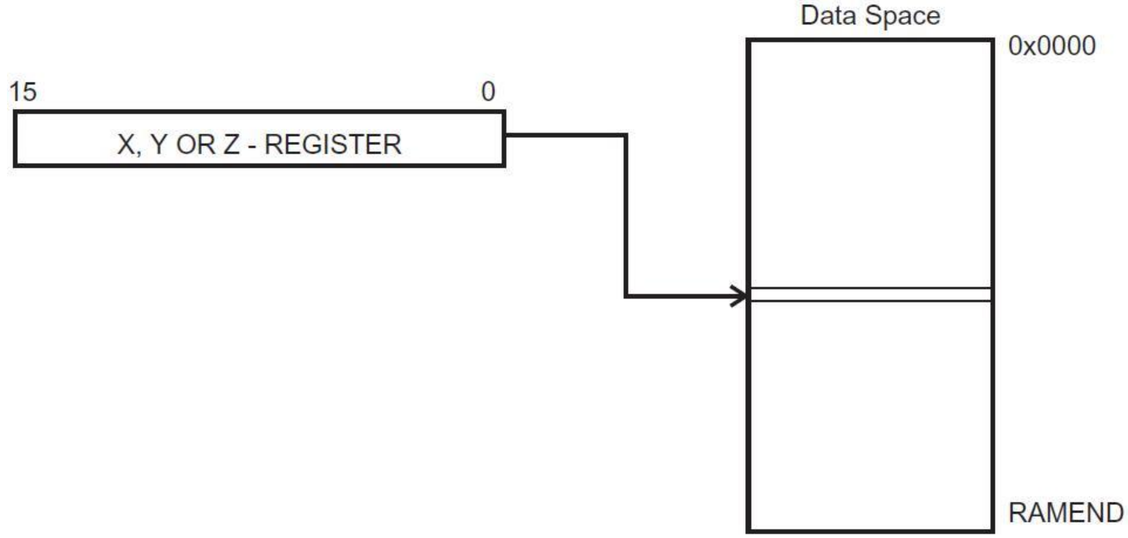
- Now let's take a closer look at how data is saved in program memory using the Define Word (.DW) Assembly Directive. For illustrative purposes we will look at a look-up table named beeHives.



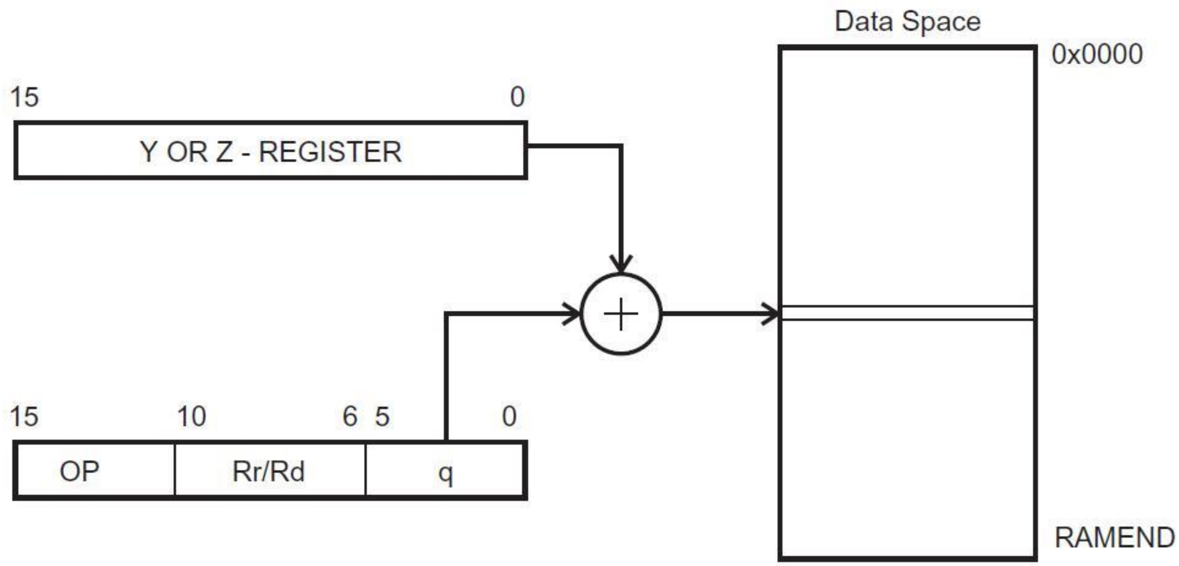
Each table entry (.DW) contains two bytes (1 16-bit word). These two bytes provide the row and column of a room containing bees. For example with respect to the maze, the room in row 00 column 04 contains 1 bee. If we look at the first entry we see it contains 0x0400. Comparing this with the corresponding Program Memory Window in AVR Studio... Wait a minute - that looks backward. From reading about the .DB assembly directive can you discover why?

DATA INDIRECT

```
ld    r16, X
st    X, r16
```

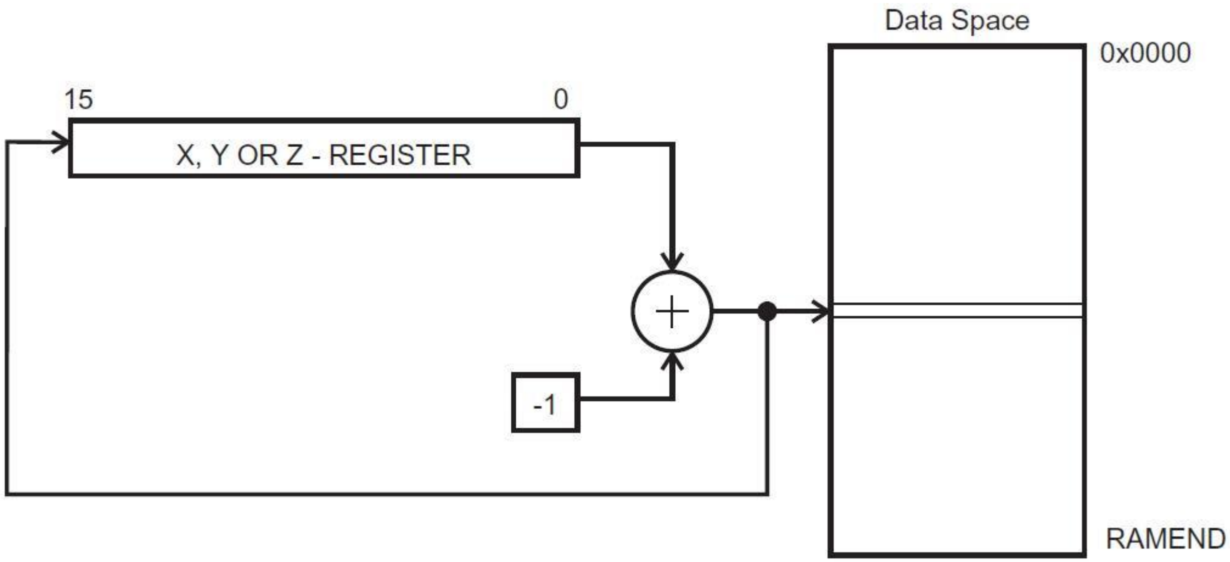



```
ldd    r4, Z+2
std    Y+2, r4
```

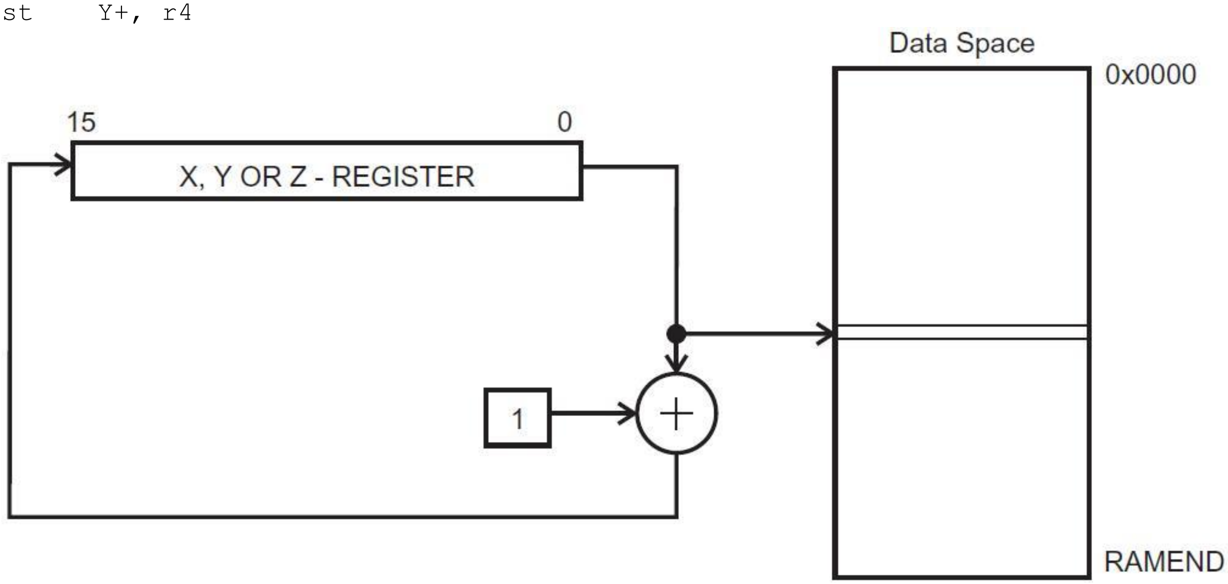


DATA INDIRECT WITH PRE-DECREMENT/POST-INCREMENT

```
ld    r16, -Y
st    -Y, r16
```




```
ld    r4, Y+
st    Y+, r4
```



DATA INDIRECT – EXAMPLE 3

- Write a program to display the 16-bit result of a 8 x 8 multiplication, where the result is stored in the r1:r0 register pair.

```
.DSEG
buffer:    .BYTE     4                 // blink status

.CSEG
.ORG 0x0000

LoadBuffer:
    ldi     XL,low(buffer)     // load address of look-up
    ldi     XH,high(buffer)
    swap    r1
    st      X+, r1
    swap    r1
    st      X+, r1
    swap    r0
    st      X+, r0
    swap    r0
    st      X+, r0
    ret

DisplayBuffer:
    ldi     XL,low(buffer)     // load address of look-up
    ldi     XH,high(buffer)
    ldi     r20, 4
cont:
    ld      r0, X+
    rcall   BCD_to_7SEG
    rcall   Delay1S
    dec     r20
    brne    cont
    ret
```