AVR ASM INTRODUCTION

Search this site

AVR ASSEMBLER TUTOR
- **1. AVR ASM BIT MANIPULATION**
- 2a. BASIC ARITHMETIC
- 2b. BASIC MATH
- 2c. LOGARITHMS
- 2z. INTEGER RATIOS for FASTER CODE
- 3a. USING THE ADC
- 3b. BUTTERFLY ADC
- 4a. USING THE EEPROM
- 4b. BUTTERFLY EEPROM
- 5. TIMER COUNTERS & PWM
- 6. BUTTERFLY LCD & JOYSTICK
- 7. BUTTERFLY SPI & AT45 DATAFLASH

Sitemap

AVR ASSEMBLER TUTOR >

1. AVR ASM BIT MANIPULATION

A MORON'S GUIDE TO BIT MANIPULATION

v1.7

by RetroDan@GMail.com

CONTENTS:

- THE "AND" OPERATIONS
- THE "OR" OPERATIONS
- THE EXCLUSIVE OR OPERATION
- THE NOT OPERATION
- THE LEFT-SHIFT OPERATIONS
- THE RIGHT SHIFT OPERATIONS
- THE "SWAP" COMMAND
- THE SBI/CBI COMMANDS
- THE SBIS/SBIC COMMANDS
- THE SBRS/SBRC COMMANDS

THE "AND" OPERATION

The "AND" operation can be demonstrated with the following circuit of two switches and a light in series:

```
Switch_1     Switch_2   LED
----/ ---------/ --------D
```

It is clear to see that the LED will only illuminate when both switches are closed to produce a complete circuit. Switch one AND switch two both have to be closed before the LED will work. This result can be displayed in a truth table where a zero means off and a one means on:

```
SW1 SW2 LED
 0   0 = 0
```

```
 0   1 = 0
 1   0 = 0
 1   1 = 1
```

The "AND" operation can be used to clear a bit to zero. From the truth table above, you can see that anything that is ANDed with a zero is zero. Lets say you wanted to clear the high bit of a register, the following code will so just that:

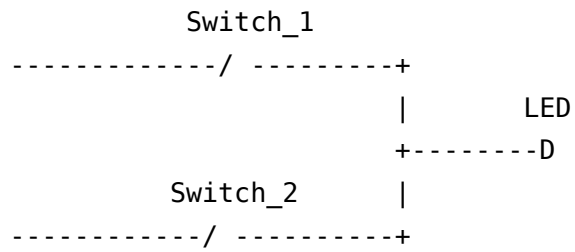
```
LDI  A,0b1111_1111     ;A = 11111111
ANDI A,0b0111_1111     ;A = 01111111
```

"AND" operations can also be used with a "bit-mask" to strip off bits we are interested in. For example if we are only interested in the highest four bits of a byte. We can use the binary number 0b1111_0000 to strip away the high nybble of that register and ignore the remainder:

```
LDI  A,0b1010_1111      ;A = 1010_1111
ANDI A,0b1111_0000      ;A = 1010_0000
```

THE "OR" OPERATION

The "OR" operation can be demonstrated with the following circuit with two switches in parallel connected to a light:



It is clear to see that the LED will light when one "OR" the other switch is closed, and even if both are closed. This can be represented by a truth table:

```
SW1 SW2 LED
 0   0 = 0
 0   1 = 1
 1   0 = 1
 1   1 = 1
```

The "OR" operation can be used to set a bit to one. From the truth table above, you can see that anything that is ORed with a one is one. Lets say we need to set the high bit of a register, the following code will do that:

```
LDI  A,0b0101_0101     ;A = 0101_0101
ORI  A,0b1000_0000     ;A = 1101_0101
```

THE EXCLUSIVE OR "EOR" OPERATION

The "EOR" operation is the same as the "OR" operation except that it is off when both switches are closed. This means the LED is on if one "OR" the other is on, but not if both are. This can be demonstrated with the following truth table:

```
SW1 SW2 LED
 0   0 = 0
 0   1 = 1
 1   0 = 1
 1   1 = 0
```

If we look at the truth table above, we will see that a one EORed with zero give a one, and a one EORed with a one gives us zero. EORing something with a one gives us the opposite or inverse. This gives us the property of flipping a bit. If you need to "blink" the high bit of a register on and off, the following code will do that without disturbing the other bits of the "A" register:

```
LDI  B,0b1000_0000
LDI  A,0b0101_0101     ;A = 0101_0101
EOR  A,B               ;A = 1101_0101
EOR  A,B               ;A = 0101_0101
EOR  A,B               ;A = 1101_0101
```

THE "NOT" OPERATION

The NOT or inverse operation means you want the opposite, ones become zero and zeros become one. The truth table for this is:

```
A      NOT_A
0        1
```

```
1        0
```

If we think back to the EOR command, we realize that when we EOR something with a one, we flip that bit. So to get the inverse or NOT of an entire value, we can EOR it with all ones:

```
LDI  B,0b1111_1111     ;ALL ONES
LDI  A,0b1010_1010     ;A=1010_1010
EOR A,B                ;A=0101_0101
```

To do this with a constant we can use the negation operator ~ (tilde):

```
LDI A,~0b1010_1010    ;A=0101_0101
```

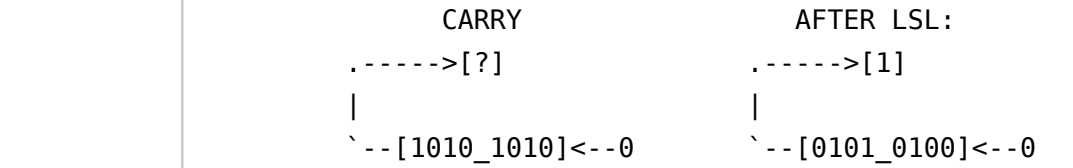
THE LEFT SHIFT OPERATIONS

The left-shift operation moves bits in a register, one place to the left. Why would we want to do this? Lets look at the example of left-shifting the value of three:

```
LDI  A,3               ;A=0b0000_0011=3
LSL  A                 ;A=0b0000_0110=3x2=6
LSL  A                 ;A=0b0000_1100=3x2x2=1
```

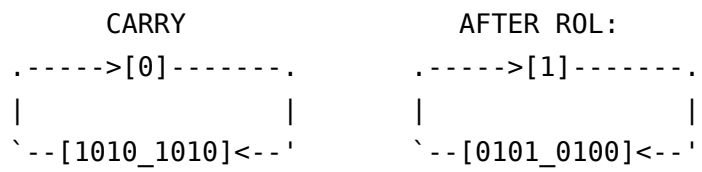
Since computers run on a Binary System, each time we shift a value one bit to the left, we are multiplying that value by two. If we want to multiply something by eight, we would left-shift it three times. What the Logical Shift Left command (LSL) does is shift the highest bit out into the carry flag, shift the contents of the register to the left one bit and shifts a zero into the lowest bit. The result is a multiplication by two:

```
CARRY    A-REGISTER
 [?]     [0101_0101]      BEFORE LEFT SHIFT V
 [0]  << [1010_1010] <<0  AFTER  LEFT SHIFT V
```



What if the highest bit was a one? If we are working with more than eight bits we can use the rotate-left command ROL which rotates the

value stored in the carry bit into the lowest bit and then loads the carry flag with the value that was shifted out the high end:



In the example above the zero that was in the carry flag is shifted into the low end of the register, the remaining bits are shifted one to the left and the high bit is shifted out and into the carry flag.

LSL always shifts a zero into the lowest bit, while ROL shifts the contents of the carry flag into the lowest bit. Using both we can multiply numbers larger than one byte.

To multiply a sixteen bit number by two, we first LSL the lower byte, then ROL the high byte, this has the net effect of "rolling" the high bit of the lower byte into the first bit of the 2nd byte. This technique can be expanded to multiply even larger numbers.

Multiplying a 32-bit number by two:

```
LSL  A1   ;MULTIPLY VALUE IN A1 BY TWO
ROL  A2   ;VALUE SHIFTED OUT OF A1 GETS SHIFT
ROL  A3   ;VALUE SHIFTED OUT OF A2 GETS SHIFT
ROL  A4   ;VALUE SHIFTED OUT OF A3 GETS SHIFT
```

RIGHT SHIFT OPERATIONS

The right-shift operation moves bits in a register, one place to the right. Why would we want to do this? Lets look at the example of right-shifting the value of twelve:

```
LDI  A,12     ;A = 0b0000_1100 = 12
LSR  A        ;A = 0b0000_0110 = 12/2=6
LSR  A        ;A = 0b0000_0011 = 12/4=3
```

Since computers run on a Binary System, each time we shift a value one bit to the right, we are dividing that value by two. If we wanted to divide something by eight, we would right-shift it three times.

What the Logical Shift Right command (LSL) does is shift a zero into the highest bit, shift the contents to the right and shifts the lowest bit into the carry flag. The result is a division by two:

```
   A-REGISTER  CARRY
   [1010_1010]  [?]  BEFORE RIGHT SHIFT VALUE
0>>[0101_0101]>>[0]  AFTER  RIGHT SHIFT VALUE
```

```
      CARRY                AFTER LSL:
       [?]<------.             [0]<------.
                 |                       |
0-->[1010_1010]---'     0-->[0101_0101]---'
```

If we are working with more than eight bits and we wish to preserve the value of the lowest bit, you can use the rotate-right command ROR which rotates the value stored in the carry bit into the highest bit and then loads the carry flag with the value that was shifted out the low end.

```
     CARRY                AFTER ROL:
.------[1]<------.       .----->[0]<------.
|               |       |                |
`->[1010_1010]---'       `->[1101_0101]---'
```

In the example above the one that was in the carry flag is shifted into the low end of the register, the remaining bits are shifted one to the right and the low bit is shifted out and into the carry flag.

LSR always shifts a zero into the highest bit, while ROR shifts the contents of the carry flag into the highest bit. Using both we can divide numbers larger than one byte.

To divide a sixteen bit number by two, you first LSR the highest byte, then ROR the lower byte, this has the net effect of "rolling" the low bit of the high byte into the high bit of the lower byte. This technique can be expanded to divide even larger numbers.

Dividing a 32-bit number by two:

```
LSR  A4   ;DIVIDE VALUE IN A4 BY TWO
ROL  A3   ;VALUE SHIFTED OUT OF A4 GETS SHIFT
ROL  A2   ;VALUE SHIFTED OUT OF A3 GETS SHIFT
```

```
ROL  A1   ;VALUE SHIFTED OUT OF A2 GETS SHIFT
```

THE "SWAP" COMMAND

The swap command exchanges the two nybbles of a byte. It exchanges the high four bits with the lower four bits:

```
[1111_0000] ---> SWAP ---> [0000_1111]

SWAP A    ;SWAPS HIGH NYBBLE IN "A" WITH LOW
```

For example if we want to isolate the high byte of register A: ANDI A,0b1111_0000 ;MASK OFF HIGH NYBBLE LSR A ;RIGHT SHIFT = /16 LSR A LSR A LSR A

Or we could use the SWAP statement as follows:

```
ANDI A,0b1111_0000
SWAP A
```

THE SBI/CBI COMMANDS

The SBI & CBI commands can be used to set or clear bits in an output port respectively. For example to set PORTB0 for output we need to set the zero-bit of the Data Direction register for Port B to one:

```
SBI DDRB,0
```

To set PORTB0 for input we need to clear bit zero:

```
CBI DDRB,0
```

The number following the register DDRB is the bit position to set or clear, it is not the value to write to the port. For example to set bit seven the command would be:

```
SBI DDRB,7     ;CORRECT
```

and not:

```
SBI DDRB,128   ;WRONG!!!
```

THE SBIS/SBIC COMMANDS

The SBIS and SBIC commands are used to branch based on if a bit is set or cleared. SBIS will skip the next command if a selected bit in a port is set, and SBIC will skip the next command if the bit is clear. For example if we are waiting for an ADC to complete, we poll the ADIF Flag of the ADCSRA register to. We could read in the register and check if the flag has been set with:

```
WAIT:  IN    A,ADCSRA       ;READ THE STATUS
       ANDI  A,0b0001_0000  ;CHECK ADIF FLAG
       BREQ WAIT
```

Or we can accomplish the same with the SBIS command that checks the status of the 4th bit without the need to read the port into a register:

```
WAIT: SBIS  ADCSRA,4
      RJMP WAIT
```

The SBIC command is identical, except it checks if a bit is zero.

THE SBRS/SBRC COMMANDS

The SBRS and SBRC commands function the same as the SBIS/SBIC commands except that they work on registers instead of ports.

Subpages (1): USING THE EEPROM

Comments

You do not have permission to add comments.