TI-83+ Z80 ASM for the Absolute Beginner

LESSON SIXTEEN:

- *Working With Bits*

WORKING WITH BITS

Bits of cornflakes? Bits of information? Bits of your sister's teddy bear or stuffed raccoon? Nope, the eight bits found in every byte.

Suppose you have a picture that you finished displaying using what you learned from Lesson 14. Let's say you want to draw a point on the screen, just a point. Remember that every bit (not byte) in plotsscreen refers to a single pixel. So you could draw the point you want by turning the appropriate bit on. The question is, how do you do that?

In the Mario game from tutorial 1, on pages 10-11, we speculated having true/false values for whether or not Mario was swimming, flying, or had the fire flower. Since each of these could be equal to 1 for true or 0 for false, we could store each of these as a bit. A review is below:

Mario is:	1 = Yes or 0 = No
A = Alive	1
B = Swimming	1
C = Flying	0
D = Has Fire Flower	1
E = Invincible	0
F = At the End of the Level	0

Using six values like this is hard to work with, and uses a lot of RAM. However, did you notice that there's only two values for each variable? In ASM, you could put these into a single variable:

1 1 0 1 0 0

Alive, Swimming, Flying, Has Fire Flower, Invincible, End of Level

Once again, the question is, if all of a sudden Mario is flying, how do we change the 3rd "0" to a "1" in order to let the calculator know that Mario is flying? Also, Mario can't fly and swim at the same time, so how do we change the 2nd "1" to a "0" to show that Mario isn't swimming?

Most of the instructions that you've learned so far, if not all of them, deal with bytes. This lesson, you'll learn instructions that deal with bits. This is a long lesson, by the way, so take your time!

When you have a number one byte long, that is, 8 bits long, the bit on the far right is bit 0. The bit on the far left is bit 7. So the 8 bits are labeled, from left to right, 7 to 0. For the Mario example, assume that Bit 7 is "Mario Alive."

	Bit 7,	6,	5,	4,	3,	2,	1,	0
%	1	1	0	1	1	1	0	0

SET Number (From 0 to 7), **One-Byte Register**

Sets a bit of the one-byte register. In other words, the bit you choose becomes 1, no matter what.

Examples: LD A, %00001000

SET 7, A ; A now equals %10001000

T-States: 8, or 15 if One-Byte Register is (HL)

Byte Storage: 2 Bytes

RES Number (From 0 to 7), **One-Byte Register**

Resets a bit of the one-byte register. In other words, the bit you choose becomes 0, no matter what.

Examples:
```
LD  D, %00001000
RES  3, D  ; D now equals %00000000
```

T-States: 8, or 15 if One-Byte Register is (HL)

Byte Storage: 2 Bytes

BIT Number (From 0 to 7), **One-Byte Register**

Checks to see if a particular bit is a one or a zero. The Z flag is set if the bit is equal to 0, and reset if the bit is equal to 1. **This shows that CP is not the only instruction where you can use JR Z, JP NZ, CALL Z, etc.**

Examples:
```
LD  E, %10000000
BIT  7, E
JR NZ, Mario_Is_Alive
```

T-States: 8, or 12 if One-Byte Register is (HL)

Byte Storage: 2 Bytes

This example program will draw a whole bunch of lines on the graph screen. There will be 48 lines, each 10 pixels long downward. Be sure to read the comments, as usual.

```
#include “ti83plus.inc”
.org $9D93
.db    t2ByteTok, tAsmCmp

        B_CALL _RunIndicOff                ;Turns off the little bar you see running at the side of the screen

        LD HL, plotsscreen          ;Start at the beginning of plotsscreen, in order to draw the lines extending
                                    ;downward.
        LD B, 0
        ;For the next part, we could always say “LD B, %10101010.  However, for the sake of practice,
        ;we’ll use the set function instead.

        SET 7,  B
        SET 5,  B
        SET 3,  B
        SET 1,  B

        LD C, 247           ;We want to draw 48 lines 10 pixels long.  To do this, we store B into HL 120 times.
                            ;You’ve learned many, many ways to do this, but we’ll use BIT this time.  If bit 7 of C
                            ;is 0, that means we are done.  That is because 247 – 120 = 127.
                            ; (127 = %01111111, but 128 = %10000000)

Loop:
        LD (hl), B          ;Now the particular set of 8 pixels will have pixel on, pixel off, pixel on, pixel off, etc.
        INC HL
        DEC C
        BIT 7, C
        JR NZ, Loop

        B_CALL _GrBufCpy
        B_CALL _getKey
        B_CALL _ClrLCDFull
        B_CALL _DispDone            ;Displays the word "Done" at the end of the program

        ret
```

While these 3 functions are convenient and excellent for messing with and retrieving individual bits, it's a little more complicated using these instructions for groups of 2 or more bits.

To illustrate this, I'm going to use the example of the Real-Time Strategy I'm coding, called Seek And Destroy. In my data, in order to save space and allow faster processing time, I use 4 bits to store certain information needed for the building's construction time. Example:

% 1 0 0 0 **0 1 0 0**

Data needed

Suppose that I put this data into register A. Putting the last four bits together, I would **want** to end up with register A being a number between 0 and 15. 0 = %0000, and 15 = %1111.

The problem is, there's 4 more bits in the number, in front of the 4 bits I want. If these 4 bits are all equal to 0, then register A will be a number between 0 and 15. BUT, if even one of these 4 bits at the beginning of register A is equal to 1, register A will be bigger than 15, since 15 = %00001111.

So how can we fix this, how do we make sure that the four bits at the front of register A are all zeros? Well, you could use the following code on the next page.

```
LD A, (Structure_Build_Time_Data)

RES 7,  A
RES 6,  A
RES 5,  A
RES 4,  A
```

That works, but it takes a lot of valuable space in your program. Furthermore, it requires a lot of valuable TIME in your program—in other words, that code is slow.

It gets worse. I want to put this number, from 0 to 15, into the last four bits of register B. (And this can be any number from 0 to 15 in the last four bits of A. We don't know what this number is.) This means, **I want the first four bits of register B to remain untouched**, in order to preserve whatever data I have in the first four bits of that register. (Even though the registers are different, this is not a hypothetical situation. I really had to do this with my program). The code on the next page can do that, but can you imagine what a nightmare it would be for speed and size?

```
Check_Bit_3_Of_A:

        BIT 3,  A
        JR  NZ,  Set_Bit_3_Of_B
        RES 3, B
        JR Check_Bit_2_Of_A

Set_Bit_3_Of_B:
        SET 3, B

Check_Bit_2_Of_A:

        BIT 2,  A
        JR  NZ,  Set_Bit_2_Of_B
        RES 2, B
        JR Check_Bit_1_Of_A

Set_Bit_2_Of_B:
        SET 2, B

Check_Bit_1_Of_A:

        BIT 1,  A
        JR  NZ,  Set_Bit_1_Of_B
        RES 1, B
        JR Check_Bit_0_Of_A

Set_Bit_1_Of_B:
        SET 1, B

Check_Bit_0_Of_A:

        BIT 0,  A
        JR  NZ,  Set_Bit_0_Of_B
        RES 0, B
        JR Continue

Set_Bit_0_Of_B:
        SET 0, B
Continue:
```

Now that you know why the 3 instructions are not so good for groups of bits, I'll show you what is.

I'm going to ask you to think back to your Ti-Basic programming. Do you recall using "and" and "or" in your "If…then" statements? Perhaps you even used Not and Xor. Let's just stick with "and" and "or" for a few minutes.

Suppose you have the following Ti-Basic program:

```
If A = 1 and B = 1

Disp "A = 1 and B = 1"
```

As you might expect, the text "A = 1 and B = 1" is only going to display if BOTH A and B are equal to 1. If A = 1 and B is some other random number, the text will not display.

What about this code?

If A = 0 or B = 0

Display “A * B = 0”

Then you have three possibilities when the text displays: Either A is equal to 0, or B = 0, or BOTH of them = 0. However, if A and B are both numbers not equal to zero, the text will not display.

How about some “fake code” such as below?

```
If Bit 7 of Register A = 1 and Bit 7 of Register B = 1

Bit 7 of Register A = 1

If Bit 6 of Register A = 1 and Bit 6 of Register B = 1

Bit 6 of Register A = 1

If Bit 5 of Register A = 1 and Bit 5 of Register B = 1

Bit 5 of Register A = 1

If Bit 4 of Register A = 1 and Bit 4 of Register B = 1

Bit 4 of Register A = 1

If Bit 3 of Register A = 1 and Bit 3 of Register B = 1

Bit 3 of Register A = 1

If Bit 2 of Register A = 1 and Bit 2 of Register B = 1

Bit 2 of Register A = 1

If Bit 1 of Register A = 1 and Bit 1 of Register B = 1

Bit 1 of Register A = 1

If Bit 0 of Register A = 1 and Bit 0 of Register B = 1

Bit 0 of Register A = 1
```

So what that means is, if you take a bit in A and compare it with the same bit in B, they both must be equal to 1. Otherwise, the bit in question in A will be 0, no matter what.

Suppose register A = %10011111 and register B = %11100001.

```
          %10011111      ;Register A
    and   %11100001      ;Register B
Result:   %10000001      ;The result is stored in register A
```

As you might guess, if we decide to use "OR" instead, then for each bit in registers A and B, only one of the two that we compare needs to be equal to 1. The particular bit in register A would be zero only if the bit in question is equal to zero in both registers A and B.

```
          %10011110
    or    %11100000
Result:   %11111110
```

With XOR, available in Z80 ASM, if we compare a bit from A with the same bit from B, the bit in question will be set to "1" in register A if the bits from A, B are not equal to each other. If they are both equal to 1 or both equal to 0, the result is "0".

```
          %10011110
    or    %11100100
Result:   %01111010
```

Exercises: Compute the result (answers on next page):

1. A = %10000101, B = %11100000. A AND B
2. A = %01010111, B = %00111000. A OR B
3. A = %01010110, B = %11001010 A XOR B
4. A = %10101010, B = %01101100 A AND B,
   A OR B,
   A XOR B

Answers:

1. %10000000
2. %01111111
3. %10011100
4. %101000, %11101110, %11000100

So what's the big deal? After all, in Z80, you can't use and, or, xor in If-Then statements. Or can you? I'll tell you what the big deal is after I give you 6 instructions.

AND **One-Byte Register**

ANDs the one-byte register with register A. The result is stored in A. Note that you can even AND register A with itself!

```
Examples:        LD  A, %00011000
                 LD C,  %11111001
                 AND  C     ; A now equals  %00011000
```

T-States: 4, or 7 if Register is (HL) One-Byte Byte Storage: 1 Byte

AND **One-Byte Value**

ANDs the one-byte value with register A. The result is stored in A.

```
Examples:        LD  A, %00011000
                 AND   %10101010  ; A now equals  %00001000
```

T-States: 7 Byte Storage: 2 Bytes

OR **One-Byte Register**

ORs the one-byte register with register A. The result is stored in A. Note that you can even OR register A with itself!

Examples:
```
LD  A, %00011000
LD H,  %11111001
OR  H       ; A now equals  %11111001
```

T-States: 4, or 7 if Register is (HL) One-Byte Byte Storage: 1 Byte

OR **One-Byte Value**

ORs the one-byte value with register A. The result is stored in A.

Examples:
```
LD  A, %00011000
OR   1; A now equals  %00011001
```

T-States: 7 Byte Storage: 2 Bytes

XOR **One-Byte Register**

XORs the one-byte register with register A. The result is stored in A. Note that you can even XOR register A with itself!

Examples: LD A, %00011000

XOR A ; A now equals 0

T-States: 4, or 7 if Register is (HL) One-Byte Byte Storage: 1 Byte

XOR **One-Byte Value**

XORs the one-byte value with register A. The result is stored in A.

Examples: LD A, %00011000

XOR %10101010 ; A now equals %10110010

T-States: 7 Byte Storage: 2 Bytes

WHAT'S THE BIG DEAL #1: We can, of course, use these values to set, reset, or even flip (0 becomes 1 and 1 becomes 0) certain bits in register A. If you want to set a bunch of bits in register A to "1", OR a number—let's say register B—to register A. Register B should have a "1" in each bit you want to set in register A. For example, if you have a number in register A, and you want to make the last 5 bits ones WITHOUT changing the values of the top 3 bits, just use the instruction OR %00011111. This means that if you have a "0" for a bit in register B, the respective bit in register A will be ignored, ignored, ignored.

If you want to **reset** a bunch of bits in register A to "0", AND a number—let's say register B—to register A. Register B should have a "0" in each bit you want to reset in register A. For example, if you have a number in register A, and you want to make the first 4 bits zeros WITHOUT changing the values of the last 4 bits, just use the instruction AND %00001111. This means that if you have a "1" for a bit in register B, the respective bit in register A will be ignored.

If you want to **flip** a bunch of bits in register A, XOR a number—let's say register B—to register A. Register B should have a "1" in each bit you want to flip in register A. For example, if you have a number in register A, and you want to flip the first bit in register A WITHOUT changing the values of the last 7 bits, just use the instruction XOR %10000000. This means that if you have a "0" for a bit in register B, the respective bit in register A will be ignored.

Exercise: What do you order if you want:

1. To make Bits 7, 5 and 1 of A equal to 1, without changing the rest of the bits?
2. To flip bit 6 of register A?
3. To make Bits 3-0 equal to 0?

Answers: 1. OR %10100010 (You can also use OR 162 ☺)

2. XOR %01000000

3. AND %11110000

WHAT'S THE BIG DEAL #2: Indeed, you can use AND, OR and XOR as If…Then instructions. Of course, you can't go "If this is true AND this is true OR this is true XOR this is true." Sad, but true. However, just like with bit, you can use AND, etc. to test and see if certain bits in a number are the way you want them.

In my S.A.D. code, I sometimes need to see if those 4 bits I mentioned previously are all zeros. To do this, I first store the number in register A, and then use the statement AND %00001111. This will clear out the top four bits. Do you know what this means? This means that if the bottom four bits are all zeros, the number itself will be zero. And of course, you know what that means: The Z flag is set!

WHAT'S THE BIG DEAL #3: OR A is the same thing as saying CP 0, and it's faster and smaller than CP 0. OR A will never, ever mess up whatever you have stored in register A, so use it whenever you can instead of CP 0.

XOR A will always, always set A equal to zero. Can you find out why? Anyways, this is smaller and faster than LD A, 0.

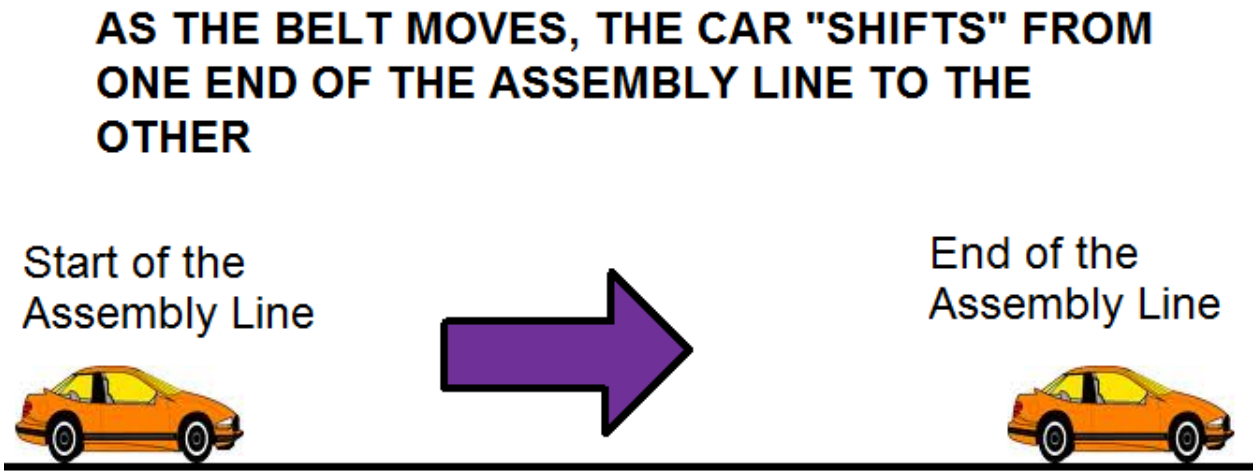
An example program is on this page. Then we have one more thing to learn in this chapter. Yes, it's a lot of information for one tutorial, but it will be worth it when you find out that next lesson will cover sprites—aka pictures you can move on the screen!

```
#include "ti83plus.inc"
.org $9D93
.db    t2ByteTok, tAsmCmp

        B_CALL _RunIndicOff
        B_CALL _ClrLCDFull

        ;The program starts by drawing a completely black screen.  Then it draws thick rectangles of alternating
        ;colors by using AND to turn off certain pixels.  Finally, we use XOR to invert the colors of the rectangles,
        ;so that instead of a black rectangle first, we have a white rectangle first.

        ld  bc, 768
        ld a, %11111111      ; All eight pixels in each byte of plotsscreen should be black
        ld hl, plotsscreen
Loop1:
        ld (hl), a
        inc hl

        ;This next set of instructions will decrease bc by 1.  We don't just say "dec bc", because if we do,
        ;no flags will be set.  We need to use the zero flag to determine if we need to loop or not.

        dec c
        jr nz, Loop1
        dec b
        jr nz, Loop1

        ;If bc is zero, the screen is completely black, so we can display the screen.

        push af                         ;Save the value of register A
        B_CALL _GrBufCpy
        B_CALL _getKey
        pop af
```

```
        and %11110000           ;Now A = %11110000, which will mean that plotsscreen contains
                                ;several black and white rectangles 4 pixels long each.
        ld bc, 768              ;Reset the counter and the location stroed in HL
        ld hl, plotsscreen
Loop2:
        ld (hl), a
        inc hl

        dec c
        jr nz, Loop2
        dec b
        jr nz, Loop2

        push af                 ;Save the value of register A
        B_CALL _GrBufCpy
        B_CALL _getKey
        pop af

        xor %11111111 ;Now A = %00001111, which will mean that plotsscreen contains
                                ;white rectangles first.
        ld bc, 768              ;Reset the counter and the location stroed in HL
        ld hl, plotsscreen
Loop3:
        ld (hl), a
        inc hl

        dec c
        jr nz, Loop3
        dec b
        jr nz, Loop3



        B_CALL _GrBufCpy
        B_CALL _getKey

        B_CALL _ClrLCDFull
        B_CALL _DispDone

        ret
```

The last thing you will learn in this lesson is how to move bits around a number. Yes, move around, like a merry-go-round or an assembly line. What do I mean by that?

Well, an assembly line moves an object, such as a car, down a belt. So if the car starts at the front, the belt will eventually move it to the end of the line. The car moves from its original position to a different position.



Yes, I do want you to remember the term "shift." The car shifted forward from its original position, in a single direction.

What about my analogy of the merry-go-round? Well, if you think about it, when you were on a merry-go-round, you started at a certain point. If you kept going, and the merry-go-round did not break down, you would eventually reach the point where you started, basically going around in circles. This is not like "shifting," where you go in a single direction and never return. Instead, you "rotate" in a circle.

Can we do this with bits? YOU BET!

RL **One-Byte Register**

Rotates the bits in the One-Byte Register once to the left. Whatever is in the carry flag is put in bit 0, and whatever is in bit 7 is put inside the carry flag.

```
Examples:          ; Suppose the carry flag is set, meaning "1"
                   LD  E, %10011000
                   RL  E   ;E now equals %00110001.
                           ;The carry flag is now set, = "1".
```

T-States: 8, or 15 if One-Byte Register is (HL) Byte Storage: 2 Bytes

RLA

This is faster and smaller than RL A, but does exactly the same thing.

```
Examples:          ; Suppose the carry flag is reset, meaning "0"
                   LD  A, %00011000
                   RLA    ;A now equals %00110000
                          ;The carry flag is now reset, = "0".
```

T-States: 4 Byte Storage: 1 Byte

RLC **One-Byte Register**

Rotates the bits in the One-Byte Register once to the left. Whatever is in bit 7 is put inside the carry flag and bit 0.

Examples:

```
LD  E, %01010101
RLC  E        ;E now equals %10101010
              ;The carry flag is now reset, = "0".
```

T-States: 8, or 15 if One-Byte Register is (HL) Byte Storage: 2 Bytes

RLCA

This is faster and smaller than RLC A, but does exactly the same thing.

Examples:

```
LD  A, %10011000
RLA    ;A now equals %00110001
       ;The carry flag is now set, = "1".
```

T-States: 4 Byte Storage: 1 Byte

For the sake of saving space, I will simply say that RR, RRA, RRC, and RRCA do the opposite of RL, RLA, RLC, and RLCA.

SLA **One-Byte Register**

Shifts the bits in the One-Byte Register once to the left. A zero is placed in bit 0, and whatever is in bit 7 is put inside the carry flag.

Examples:

```
LD  D, %10011000
SLA  D       ;D now equals %00110000
             ;The carry flag is now set, = "1".
```

T-States: 8, or 15 if One-Byte Register is (HL) Byte Storage: 2 Bytes

SRL **One-Byte Register**

Shifts the bits in the One-Byte Register once to the left. A zero is placed in bit 7, and whatever is in bit 0 is put inside the carry flag.

Examples:

```
LD  B, %10011000
SRL  B       ;B now equals %01001100
             ;The carry flag is now reset, = "0".
```

T-States: 8, or 15 if One-Byte Register is (HL) Byte Storage: 2 Bytes

There are many purposes for shifting and rotating bits. Two of the most common are for drawing sprites and for multiplying/dividing by powers of two (meaning diving by 2, 4, 8, 16, 32, 64, etc.) We will learn how to use shifting/rotating for sprites next lesson. The next example program will demonstrate multiplication and division by powers of two. But to understand the concept, try the following 6 math problems, and feel free to use a calculator:

1. 2000 / 10
2. 2000 / 100
3. 2000 / 1000
4. 2*10
5. 2*100
6. 2*1000

You will notice that in problem one, the answer was 200: what you did was shift the "2" one place to the right! And when you divided by 100 (meaning 10 * 10), you shifted the "2" TWO places to the right! So as you can probably guess, when you divide by 10 * 10 * 10, you shift three places to the right.

When you multiply by 10, you shift 1 place to the LEFT. So when you multiply by 10 * 10 * 10, you shift 3 places to the left.

However, this is when you're dealing with decimal numbers. In binary numbers, multiplying by 2 will shift one place to the left, multiplying by 2 * 2 will shift **two** places to the left, dividing by 2 * 2 * 2 will shift 3 places to the RIGHT, and so on and so forth.

```
#include “ti83plus.inc”
.org $9D93
.db    t2ByteTok, tAsmCmp

        B_CALL _RunIndicOff
        B_CALL _ClrLCDFull

        ld  a,  5

        ;We will now compute 5 * 16.

        sla a \ sla a \ sla a \ sla a \ sla a        ;Use a backslash to put multiple commands on the same line.

        ld  h, 0
        ld  l, a
        B_CALL _DispHL
        B_CALL _getKey
        B_CALL _ClrLCDFull

        ld a, 128

        ;We will now compute 128 divided by 64.

        srl a \ srl a \ srl a
        srl a \ srl a \ srl a

        ld  h, 0
        ld  l, a
        B_CALL _DispHL
        B_CALL _getKey
        B_CALL _ClrLCDFull
        ret
```