TI-83+ Z80 ASM for the Absolute Beginner

LESSON SEVEN:

- *General Purpose One-Byte Registers*
- *The B Register*
- *New Coding Instructions*

GENERAL PURPOSE ONE-BYTE REGISTERS

Quick review, because the more you remember the purpose of registers, the better. Registers are used as a CPU's temporary memory. The CPU cannot solve a problem in RAM; or when it can, RAM is very slow. So the CPU uses registers as temporary memory to help in solving a problem. However, registers are not good for long term memory, just for holding values long enough for the CPU to finish its task.

If you remember Lesson Four, you remember that registers H and L can be combined into HL. Next lesson, you will learn the importance of HL and its special purpose. However, when H and L are used by themselves, you can use them for almost anything that the calculator needs them for in terms of "working memory." For example, suppose you want to add two numbers from RAM…one of the numbers is found in RAM at address 36864, and the other is found in RAM at address 36965.

```
        ld a, (36864)

; Notice that A is full.  To get the second value from RAM address 36865,
; we need to make sure A is saved.  Otherwise, if we let a be equal to
; whatever is in 36865, we lose the value we got from RAM address 36864.
; Since register h is empty, we put the value from register a into register h.

        ld h, a   ; Now we have saved the value from a to h.
        ld a, (36865)
        add a, h
```

In this example, we needed another register to hold a value so we could add it later. This is just one of the many examples of using h and l for register memory when "working memory" is needed.

However, H and L are registers that, by themselves, have no special purpose, something that they alone are good for. While register A is used for math, H and L are, when used by themselves, general-purpose registers. You use them when you need registers for the CPU to solve a problem and some other registers are tied up with values that you don't want to lose. And you cannot use them for math like register A.

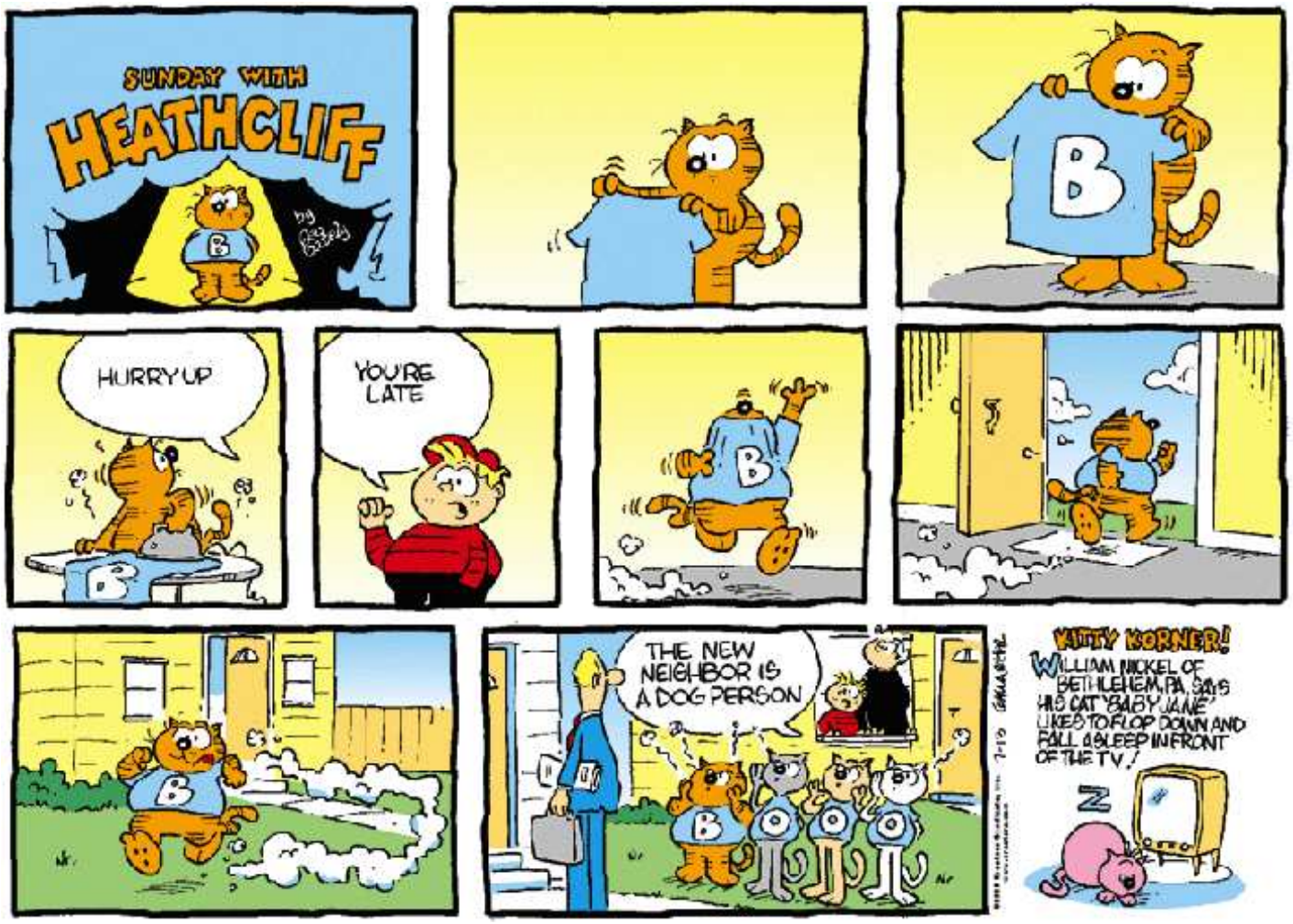
By the way, although A is used for math, you can also use it for the same purpose you use H and L for. That is, if A is empty and you need to hold a temporary value when other registers are being used, you can use A to hold the value that you need.

What if registers H, A and L are all tied up? D and E are two more registers available. Like H and L, D and E have a special purpose when used together (in other words, DE), but they do not have a special purpose when used by themselves.

And if H, A, L, D and E are tied up? Well, there's always two more registers, B and C. And like H, L, D and E, B and C have a special purpose when used together. HOWEVER, registers B and C are like register A: they have a special purpose, something they can do that no other registers can do! We'll learn the special purpose of C in another lesson. The special purpose of B will be covered in a moment. Before that, I want you to be aware that there are other one-byte registers, which we will look at in future lessons.

Remember, these registers are one byte. If you try to go over 255, the register will reset itself to zero. If you try to go below zero, the register will reset itself to 255.

THE B REGISTER



Although register B can be used to hold values when other registers are tied up, register B is also commonly used as a counter in terms of a Do-While or a For loop. It tells your ASM program how many times you want to loop a section of code. It is much, much faster than using the **equivalent** of a Ti-Basic While…End or a For…Loop, but if you use it in its purest form, it has limitations.

If you have a loop you need to execute/run a certain number of times, the B register allows you to do this kind of thing, and in an efficient manner. You let B be equal to the number of times you want to run a loop. Then for each time the loop runs, B is decreased by 1. If B does not equal zero, the loop continues.

However, "be" (no pun intended) aware of the following:

1. Register B can only tell the calculator how many times left to loop. Therefore, when Register B is used for this special While…End purpose, you can only decrease B. You cannot increase B.

2. Register B, when used for this special purpose, can only be decreased by 1. Remember that B is "how many times to loop," so you can't decrease it by a value other than one.

3. DJNZ cannot jump very far. It can only jump as far as JR can.

To use register B for performing loops, set B to the number of times you want to loop. Use a label at the beginning of your loop, and use the instruction DJNZ label at the end of your loop. When DJNZ is reached, B is decreased by 1. If B is not zero, it will jump to your label, aka the beginning of the loop. If B is zero, the program continues.

```
#include “ti83plus.inc”
.org  40339
.db    t2ByteTok, tAsmCmp

        B_CALL _ClrLCDFull

        ld b, 200  ; We want to loop 200 times
        ld a, 5

Beginning_Of_Loop:

        add a, 1

        djnz Beginning_Of_Loop

        ; Since a was increased by one 200 times, a should equal 205.

        ld h, 0
        ld l, a

        B_CALL _DispHL
        B_CALL _getKey
        B_CALL _ClrLCDFull

        ret
```

Exercises:

Here are three programs to loop several times. Each results in a numerical answer that is displayed with _DispHL. Write programs to carry out these computations. Answers will be displayed on the next page, although portions such as “ret” and “.org 40339” will be left out of the answers. Use any names you want for labels, and anything you want for comments.

1. Start with register A equal to 250. Then subtract 2 from register A 100 times. Remember that sub number subtracts the number from register A. For example, sub 1 means subtract 1 from register A.
2. Let register H be equal to 5, and start with register A equal to zero. Add H to register A 10 times.
3. Let register H equal 2, register L equal 3, register D equal 4, and register E equal 5. Start with A equal to 6. Add H, L, D and E to A 5 times.

ANSWER TO ONE:

```
        B_CALL _ClrLCDFull

        ld b, 100  ; We want to loop 100 times
        ld a, 250

Beginning_Of_Loop:

        sub 2

        djnz Beginning_Of_Loop

        ; Since a was decreased by two 100 times, a should equal 50.

        ld h, 0
        ld l, a

        B_CALL _DispHL
```

ANSWER TO TWO:

```
        B_CALL _ClrLCDFull

        ld b, 10  ; We want to loop 10 times
        ld a, 0
        ld h, 5

Loop:

        add a, h

        djnz Loop

        ; Since a was increased by five 10 times, a should equal 50.

        ld h, 0
        ld l, a

        B_CALL _DispHL
```

ANSWER TO THREE:

```
        B_CALL _ClrLCDFull

        ld b, 5  ; We want to loop 10 times
        ld a, 6
        ld h, 2
        ld l, 3
        ld d, 4
        ld e, 5

Loop:

        add a, h
        add a, l
        add a, d
        add a, e

        djnz Loop

        ; Since a was increased by fourteen 5 times, a should equal 76.

        ld h, 0
        ld l, a

        B_CALL _DispHL
```

By the way, notice that in problems 2 and 3, H is wiped out and replaced with another value, 0. Remember that registers are not meant to hold permanent values, just temporary values.

NEW CODING INSTRUCTIONS

Now that we have more registers to play with, here are some more instructions, as well as a review of the previous instructions you learned. Since you know the Ti-Basic language, I'm assuming you know what parameters are.

If you see a parameter called **One-Byte Register**, you can use a register A, B, C, D, E, H, or L in the parameter. Any other one-byte registers you learn about CANNOT be used in this parameter.

If you see a parameter called **Label**, you can use a label. OR, since you understand that Labels simply refer to RAM addresses, you can use a number to refer to a RAM address.

If you see a parameter called **Variable**, you can use any variable in your parameter. OR, since you understand that variables simply refer to RAM addresses (just like labels), you can use a number to refer to a RAM address.

If you see a parameter called **One-Byte Value**, you use an 8-Bit value in the parameter, any 1-Byte number. This, of course, means a number from 0 to 255.

Here are a couple of things you need to know: Each instruction will come with a byte-storage, and a T-State value. Byte-storage is how many bytes the instruction will take when translated into numbers, the language of the calculator. The more byte-storage you have, the bigger your ASM program will be. T-State is how fast the routine will run, on any Ti-83+ or Ti-84+ calculator. The smaller the T-State value is, the faster the routine is. A Ti-83+ (NOT the Silver Edition or the Ti-84+) executes 6,000,000 T-States per second, and the Silver Edition/Ti-84+ can execute up to 15,000,000 T-States per second. Keeping this in

mind, part of your goal as an ASM programmer is to find the fewest number of T-States that gets the job done. An instruction can have as few as 4 T-States, or as many as 23 or more T-States. The fewer T-States you have, the more your calculator can do per second. This means that on a regular Ti-83+, if you want your game to run at 30 frames per second, you can have no more than 200,000 T-States for each of your 30 loops. If you want your game to run at 60 frames per second, you can have no more than 100,000 T-States for each of your 60 loops. Keeping track of every single byte and T-State is a useless waste of time, but keeping the general idea in your head can optimize your program by as much as 75% or more—I'm dead serious.

One more thing, parentheses are required where indicated. Remember that parentheses around a ram address means you want to access the data inside that particular location in ram.

LD **One-Byte Register**, **One-Byte Register**

Stores any value from one register into another register. The first register is what to store to, and the second register is the register to store FROM.

```
Examples:       LD  A,   H    ; A now holds the value H is equal to
                LD  B,   C
```

T-States: 4 Byte Storage: 1 Byte

LD **One-Byte Register**, **One-Byte Value**

Stores any value, up to 255, into a one-byte register.

Examples: LD L, 55

LD D, 127

T-States: 7 Byte Storage: 2 Bytes

ADD A, **One-Byte Value**

Adds any value, up to 255, into register A

Examples: ADD A, 122

T-States: 7 Byte Storage: 2 Bytes

ADD A, **One-Byte Register**

Adds the value inside a One-Byte Register into register A. You can even add A to itself! (Effectively doubling the value)

Examples: ADD A, H

ADD A, A

T-States: 4 Byte Storage: 1

SUB **One-Byte Value**

Calculates A minus any one-byte value, and stores the answer in A.

Examples: SUB 97

SUB 212

T-States: 7 Byte Storage: 2

SUB **One-Byte Register**

Calculates register A minus the value inside a One-Byte Register, and stores the answer in A. You can subtract A from itself, although this will simply mean A is equal to 0.

Examples: SUB H

SUB C

T-States: 4 Byte Storage: 1

DEC **One-Byte Register**

INC **One-Byte Register**

DEC subtracts 1 from a one-byte register. INC adds 1 to a one-byte register.

Examples:
```
INC  H
DEC  B
```

T-States: 4 Byte Storage: 1

JR **Label**

Jumps to a label. The label can only be a short distance away. Spasm will tell you if the label is too far away to jr to.

Examples:
```
JR   Add_One_To_A
```

T-States: 12 Byte Storage: 1

JP **Label**

Jumps to a label. When you use JP instead of JR, you can jump much further in your program; in fact, you can jump anywhere in your program.

Examples: JP Subtract_One_From_B

T-States: 12 Byte Storage: 1

LD A, (**Variable**)

Retrieves the value stored inside Variable, and puts it in register A. Recall that the variable pertains to a particular RAM address, so you're accessing the value stored in RAM.

Examples: LD A, (Is_Game_Over)

T-States: 13 Byte Storage: 3

LD (**Variable**), A

Stores whatever is inside of A into a Variable.

Examples: LD A, 5

LD (Number_Of_Lives), A

T-States: 13 Byte Storage: 3

DJNZ **Label**

Used for looping, where register B is equal to the number of times you want to loop. When DJNZ is reached, B is decreased by 1. Afterwards, if B does not equal zero, the program jumps to the label you specified.

Examples: DJNZ Beginning_Of_Loop

T-States: 8 if B is equal to 0 Byte Storage: 2

13 if B is not equal to 0

I'll give you three programs to try out. Then try some of your own using these instructions.

But before I do that, I want to get you excited about the next lesson. Are you ready to learn about if…then…else? Of course you are. But to understand this, you need to understand about a one-byte register called F. So next lesson, I'll talk about register F, and you'll learn how to use it to create if…then…else code. Oh, and you'll also learn how to display text!

PROGRAM 1: A basic multiplication program. Since multiplication in its most basic form is repeated addition, this is a basic, but therefore un-optimized, program that repeats addition. Be sure to remember that if the product is greater than 255, you'll get a very weird answer

```
#include "ti83plus.inc"
.org  40339
.db    t2ByteTok, tAsmCmp

        B_CALL _ClrLCDFull

;  Use variables  Factor1 and Factor2 to hold the numbers you want to
multiply.

        ld a, (Factor1)
        ld h, a
        ld a, (Factor2)
        ld b, a
        ld a, 0                 ;We want register A to be empty.

Multiply:

        add a, h
        djnz Multiply

        ld h, 0
        ld l, a

        B_CALL _DispHL
        B_CALL _getKey
        B_CALL _ClrLCDFull

        ret

Factor1:
        .db 10

Factor2:
        .db 15
```

PROGRAM 2: An addition program, but this time, it will display the two numbers you add as well as the final answer.

```
#include "ti83plus.inc"
.org  40339
.db    t2ByteTok, tAsmCmp

        B_CALL _ClrLCDFull

;  Use variables  Addend1/Addend2 to hold the numbers you want to add

        ld a, (Addend1)
        ld b, a                 ; Don't use any other registers.  Remember that
                                ; the calculator uses registers to carry out a task.
                                ; When you run B_CALL _DispHL, registers
                                ; A, D, E, H and L are destroyed since they are
                                ; used during _DispHL.
        ld a, (Addend2)
        ld c, a
        ld a, 0                 ;We want register A to be empty.

Add:                    ; The label is not really necessary, it's for reference

        add a, b
        add a, c

        ld h, 0
        ld l, b

        ld b, a                 ; We will lose the value in A when _DispHL is
                                ; used.  Since we don't need register B anymore,
                                ; we use it to hold the sum from register A so that
                                ; we don't lose it.

        B_CALL _DispHL
        B_CALL _getKey

        ld h, 0
        ld l, c

;The program is continued on the next page
```

```
        B_CALL _DispHL
        B_CALL _getKey

        ld h, 0
        ld l, b                 ; The sum of our two addends.

        B_CALL _DispHL
        B_CALL _getKey
        B_CALL _ClrLCDFull

        ret

Addend1:
        .db 20

Addend2:
        .db 35
```

PROGRAM THREE: Solves the program (15-1) + (14+3). This is not the best way to solve this problem, but it does demonstrate how to use INC and DEC.

```
#include “ti83plus.inc”
.org  40339
.db    t2ByteTok, tAsmCmp

        B_CALL _ClrLCDFull

        ld d, 15
        dec d
        ld e, 14
        inc e
        inc e
        inc e

        ld a, 0
        add a, d
        add a, e

        ld h, 0
        ld l, a

        B_CALL _DispHL
        B_CALL _getKey
        B_CALL _ClrLCDFull

        ret
```