TI-83+ Z80 ASM for the Absolute Beginner

LESSON FIVE:

- *The Translation of the 1 + 5 Program*
- *Labels, Variables, and How the Calculator Works with Them*

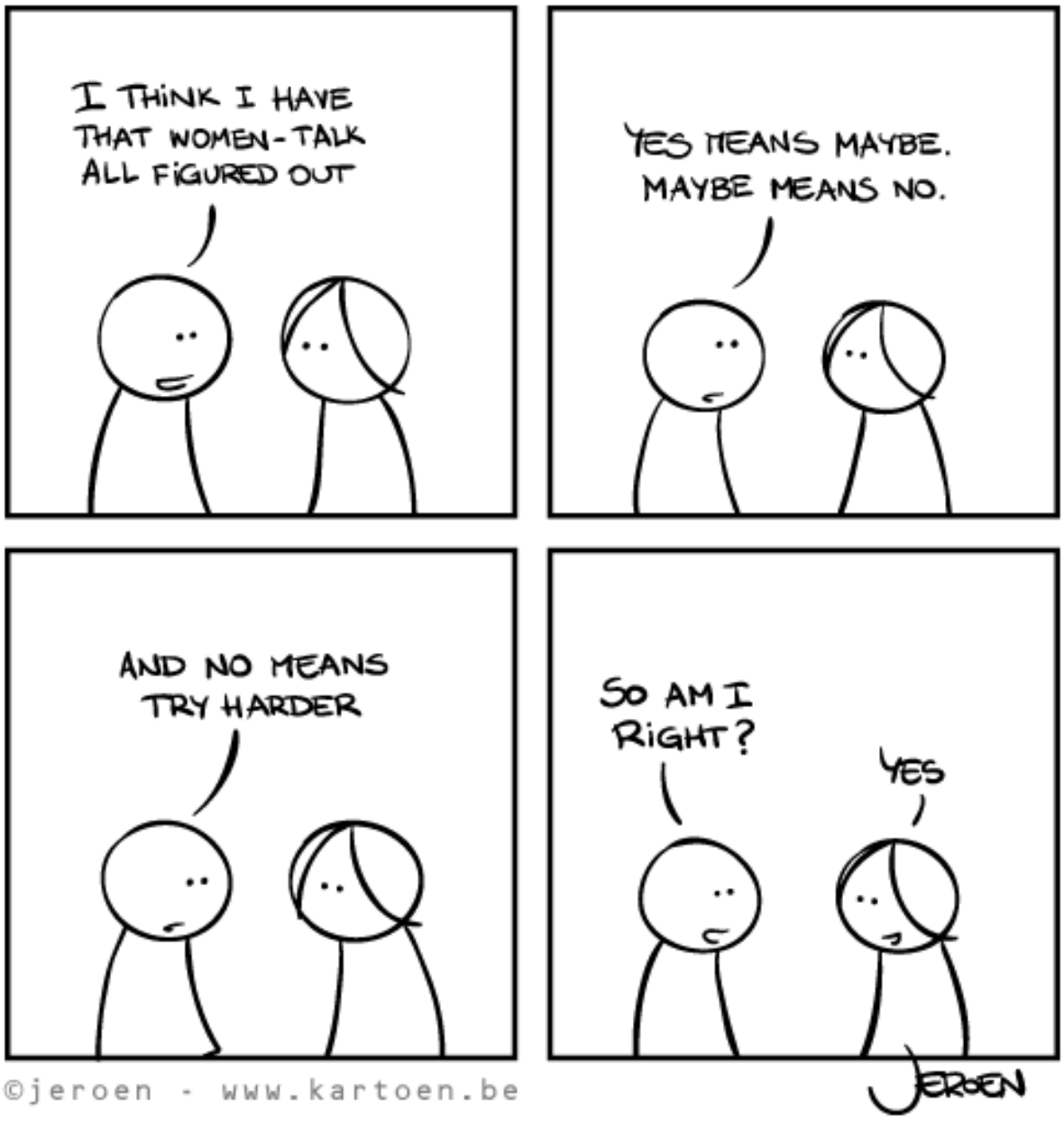
THE TRANSLATION OF THE 1 + 5 PROGRAM

I would like you to remember, for this lesson, that when you write an ASM program, it is translated into a bunch of numbers that the calculator reads, and the calculator reads these numbers and translates them into instructions to execute. Feel free to review Lesson #1 if you need to. Also remember that the program is stored in ram starting at RAM address 40339.

Most ASM tutorials do not tell you what this translated program looks like. While I don't argue the methods used in those tutorials, I don't agree that it's unnecessary to understand the translation of an ASM program. It's **extremely easy** to see the translation of an ASM program, and it's one of your first steps to see HOW something works. And believe me: if you understand how your program works and why, you get awesome calculator games like Desolate and Wolfenstein. Therefore, throughout this lesson, and perhaps from time to time on other lessons, I'll show you some ASM programs translated into numbers.

Like I said, this part is not difficult. Just remember that these lessons are for the absolute beginner, so I'm putting this up knowing you as the reader can handle it. Besides, unlike a the translation of a simple Visual Basic program or even a simple C++ program, the translation of a simple Ti-83+ ASM program takes only, oh, maybe 25-30 bytes.

The translation for the first ASM program you wrote is on page 4. Some of the translations I will explain immediately, others I will explain later, and some of the translations do not require explanation. By the way, a blue box means a number that takes two bytes, aka a word.


I THINK I HAVE THAT WOMEN-TALK ALL FIGURED OUT
YES MEANS MAYBE. MAYBE MEANS NO.
AND NO MEANS TRY HARDER
SO AM I RIGHT?
YES
©jeroen - www.kartoen.be
JEROEN

ASM	RAM ADDRESS ON CALCULATOR	TRANSLATION, IN DECIMAL NUMBERS
#include “ti83plus.inc”		
.org 40339		
.db t2ByteTok, tAsmCmp	40339	187 109
B_CALL _ClrLCDFull	40341	239 17728
ld a, 1	40344	62 1
add a, 5	40346	198 5
ld h, 0	40348	38 0
ld l, a	40350	111
B_CALL _DispHL	40351	239 17671
B_CALL _getKey	40354	239 18802
B_CALL _ClrLCDFull	40357	239 17728
ret	40360	201

For starters, notice that the first two lines aren’t translated. The first line, which tells the calculator to include the file “ti83plus.inc,” is not translated, because it contains code that is only put in the program when needed, and it tells SPASM exactly how to put it in the calculator. In other words, SPASM puts in the program what is needed, when needed, from ti83plus.inc…SPASM does not need to put in the whole file.

I will explain later in this lesson why .org 40339 is not translated/compiled, but for right now, the main reason is the line is a reference for SPASM, not for the calculator.

_ClrLCDFull, _getKey, _DispHL, t2ByteTok and tAsmCmp are all constants. Since _ClrLCDFull is a constant equal to 17728, SPASM replaces _ClrLCDFull with the number 17728 wherever _ClrLCDFull is located. Similarly, t2ByteTok is a constant equal to 187, so SPASM saw t2ByteTok and translated it to the number 187.

_ClrLCDFull, _getKey, and _DispHL are constants referring to where these instructions needed by the ASM program are located. They are ROM addresses, not RAM addresses. The instructions that ASM programs can access are located in ROM as part of the Ti-83+ operating system.

Notice that when you choose a value to load into A or H, the translation is equal to that value? Pretty sweet, huh? But not surprising.

Now to tell you about .db. .Db is used to enter data for the calculator. When you use .db, you're entering exact data that you want, so SPASM does nothing to alter the data. For instance, if you type in .db 1, 2, 3, 4, 5 and 6, the translated program contains, in decimal numbers, 1 2 3 4 5 6 in that order: nothing altered, nothing changed. All ASM programs require the numbers 187 and 109 to identify them as ASM programs, so .db tells SPASM that these numbers should be in the translated program EXACTLY as the numbers 187 and 109.

However, there is one exception to this rule, which you'll learn about in a later lesson.

Exercise: Translate the following program into numbers. The answer is on the next page.

Hint: Remember that Number_Seven and Number_Four are constants, since they use the word .equ. What does Spasm do with constants?

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

NumberSeven .equ 7
NumberFour .equ 4

        B_CALL  _ClrLCDFull
        ld a, NumberSeven

; Solve the problem 7 + 4

        add a, NumberFour
        ld h,0
        ld l, a

        B_CALL _DispHL
        B_CALL _getKey
        B_CALL  _ClrLCDFull
        ret
```

ANSWER:

ASM	RAM ADDRESS ON CALCULATOR	TRANSLATION, IN DECIMAL NUMBERS
#include “ti83plus.inc”		
.org 40339		
NumberSeven .equ 7		
NumberFour .equ 4		
.db t2ByteTok, tAsmCmp	40339	187 109
B_CALL _ClrLCDFull	40341	239 17728
ld a, NumberSeven	40344	62 7
add a, NumberFour	40346	198 4
ld h, 0	40348	38 0
ld l, a	40350	111
B_CALL _DispHL	40351	239 17671
B_CALL _getKey	40354	239 18802
B_CALL _ClrLCDFull	40357	239 17728
ret	40360	201

LABELS, VARIABLES, AND HOW THE CALCULATOR WORKS WITH THEM

As a Ti-Basic programmer, you understand that Labels are used to mark places where your program needs to jump to from time to time. You also understand that Variables are used to hold values that you need saved. But do you ever get annoyed that Labels can only be two letters/numbers long and variables are also somewhat limited?

In ASM programming, names for labels and variables can be as big as you want. However, there are two rules that apply to SPASM when applying variable names:

1. Labels and variables cannot start with a number
2. Labels and variables can only contain numbers, letters and underscores.

As a good rule of thumb, you should include a colon at the end of your labels and variables. It is not required, but it is good practice. Here are some examples of valid labels:

FunctionNumberOne:

Function_Number_One:

Function_Number1:

And, here are some examples of invalid labels and variables:

2Function1:

Function#1:

We'll start by working with labels. You'll learn later how to tell the calculator when to jump and when not to jump, but for right now, I'll be teaching you how to tell the calculator to "goto" your label no matter what. There are two ways to do this, and it depends on how far away you want to jump.

JR is an abbreviation for "Relative jump." You can use JR whenever the distance to where you want to jump is short. Spasm will tell you when you cannot use JR by saying "Relative jump is over 128 bytes." In other words, when Spasm is translating your program, if the label is more than 128 bytes away, you cannot use JR.

In which case you use JP, which is a "jump." This allows much bigger jumps. In fact, the jumps allowed are so big that you can even jump outside your program.

There's a big, big difference between JR and JP which will be described later in this lesson. However, some smaller differences are, JR takes fewer bytes and, for the most part, is faster than JP. You should use JR every time you need a jump; if you can't use it, Spasm will tell you, and you can change it to JP in a matter of moments.

Here's an ASM program for you to write. By the way, if you decide to just copy and paste the program (which I don't recommend), at least rewrite the first #include line. If you just copy and paste it, chances are Spasm won't work correctly.

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


        B_CALL  _ClrLCDFull
        ld a, 2
; Solve the problem 2 + 5
        add a, 5
        ret
        ld h,0
        ld l, a


        B_CALL _DispHL
        B_CALL _getKey
        B_CALL  _ClrLCDFull
        ret
```

What happened here? Notice the **ret** after add a, 5? The program ended early! Now, don't give me crud such as "Oh, we can easily fix that by just removing the ret statement!" That is very true, but just bear with me. We're going to take a different approach. Take this program,

and add the lines highlighted in blue. Then run it, and it should work correctly:

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


        B_CALL  _ClrLCDFull
        ld a, 2
; Solve the problem 2 + 5
        add a, 5
        jr Continue_Program
        ret
Continue_Program:
        ld h,0
        ld l, a


        B_CALL _DispHL
        B_CALL _getKey
        B_CALL  _ClrLCDFull
        ret
```

Now, I would like to show you how THIS program translates so you can see how JR works. This is important, believe me.

ASM	RAM ADDRESS ON CALCULATOR	TRANSLATION, IN DECIMAL NUMBERS
#include “ti83plus.inc”		
.org 40339		
.db t2ByteTok, tAsmCmp	40339	187 109
B_CALL _ClrLCDFull	40341	239 17728
ld a, 2	40344	62 2
add a, 5	40346	198 5
jr Continue_Program	40348	24 1
ret	40350	201
Continue_Program:		
ld h, 0	40351	38 0
ld l, a	40353	111
B_CALL _DispHL	40355	239 17671
B_CALL _getKey	40357	239 18802
B_CALL _ClrLCDFull	40360	239 17728
ret	40363	201

Notice that the label is not translated! The calculator uses a different approach than what you might expect. If you notice, ret is only one byte of instructions when translated. So, jr tells the calculator to skip one byte of instructions, hence the number “1” in the translation.

The program “jumps”, “skips” over one byte of instructions. Since ret is only one byte of instructions when translated, ret is skipped.

Let’s replace jr with jp. Then notice there’s a change:

ASM	RAM ADDRESS ON CALCULATOR	TRANSLATION, IN DECIMAL NUMBERS
#include “ti83plus.inc”		
.org 40339		
.db t2ByteTok, tAsmCmp	40339	187 109
B_CALL _ClrLCDFull	40341	239 17728
ld a, 2	40344	62 2
add a, 5	40346	198 5
jp Continue_Program	40348	195 40352
ret	40351	201
Continue_Program:		
ld h, 0	40352	38 0
ld l, a	40354	111
B_CALL _DispHL	40356	239 17671
B_CALL _getKey	40358	239 18802
B_CALL _ClrLCDFull	40361	239 17728
ret	40364	201

This time, the jump is not translated as "how far to jump." It is translated as "**where** to jump." Instead of being told to skip a number of bytes, the calculator is told the RAM address to jump to.

Now, I can tell you why .org is not translated, but why it is needed as a reference for Spasm. As was stated earlier, the ASM program is run from RAM address 40339 of the calculator. However, Spasm doesn't know that. It assumes that the address starts at 0! So suppose you did not tell Spasm that the calculator starts the program at 40339. Then, when jp is translated, it would read as 195 13, meaning it would tell the calculator to jump to address 13, **not** address 40352. Your program would not work. Try it! Remove the line .org 40339, and run the program on an emulator. The answer to the problem 2 + 5 is not displayed.

Exercise:

Before the label Continue_Program:, add three more "rets." For example:

```
jr Continue_Program
ret
ret
ret
ret
Continue_Program:
```

Now translate the program using jp and the program using jr. Answers are on the next pages: jr on the first page, and jp on the second.

ANSWER FOR JR:

ASM	RAM ADDRESS ON CALCULATOR	TRANSLATION, IN DECIMAL NUMBERS
#include “ti83plus.inc”		
.org 40339		
.db t2ByteTok, tAsmCmp	40339	187 109
B_CALL _ClrLCDFull	40341	239 17728
ld a, 2	40344	62 2
add a, 5	40346	198 5
jr Continue_Program	40348	24 4
ret	40350	201
ret	40351	201
ret	40352	201
ret	40353	201
Continue_Program:		
ld h, 0	40354	38 0
ld l, a	40356	111
B_CALL _DispHL	40358	239 17671
B_CALL _getKey	40360	239 18802
B_CALL _ClrLCDFull	40363	239 17728
ret	40366	201

ANSWER FOR JP:

ASM	RAM ADDRESS ON CALCULATOR	TRANSLATION, IN DECIMAL NUMBERS
#include “ti83plus.inc”		
.org 40339		
.db t2ByteTok, tAsmCmp	40339	187 109
B_CALL _ClrLCDFull	40341	239 17728
ld a, 2	40344	62 2
add a, 5	40346	198 5
jp Continue_Program	40348	195 40354
ret	40350	201
ret	40351	201
ret	40352	201
ret	40353	201
Continue_Program:		
ld h, 0	40354	38 0
ld l, a	40356	111
B_CALL _DispHL	40358	239 17671
B_CALL _getKey	40360	239 18802
B_CALL _ClrLCDFull	40363	239 17728
ret	40366	201

For the last part of this lesson, we will talk about variables. Variables are, of course, stored in RAM to retain more permanent storage. Thus, variables can be stored anywhere in RAM that is safe to use. For now, since your ASM program is stored in RAM, you will store variables as part of the program. Since your variable is part of the program while it is running, nothing except your program can access that part of RAM, so your variable is safe if you do not mess with it carelessly.

To store a variable, you use a combination of a label and .db. Using .db, you specify the value you wish the variable to hold at the beginning of the program. This is usually zero. However, let's use an example of a number of lives being equal to 5.

```
Number_Of_Lives:
    .db 5
```

To understand how to access this, remember that the variable is stored in RAM. Therefore, you access the variable using a RAM address. Just like labels pertain to a particular RAM address (and jp tells the calculator to jump to that particular RAM address), variable names pertain to particular RAM addresses. I will tell you about accessing the variable in a moment, but it is very important you remember that the value 5 is stored in RAM and accessed by the address where it is located. On the next page is a small program, and its translation:

ASM	RAM ADDRESS ON CALCULATOR	TRANSLATION, IN DECIMAL NUMBERS
#include “ti83plus.inc”		
.org 40339		
.db t2ByteTok, tAsmCmp	40339	187 109
B_CALL _ClrLCDFull	40341	239 17728
ret	40343	201
Number_Of_Lives:		
.db 5	40344	5

So Number_Of_Lives pertains to RAM address 40344. Inside 40344 is the number of lives.

There are many, many different ways to access this value. For now, use the statement ld a, (variable name). You always use parentheses when you want to access whatever is inside of RAM at a particular address. So by using the statement ld a, (Number_Of_Lives), you are telling the calculator to access whatever data is in RAM address 40344. Ld a, (Number_Of_Lives) is the same as ld a, (40344). So now a contains the value 5.

On the next page is a program that demonstrates this.

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

        B_CALL  _ClrLCDFull
        ld a, (Number_Two)
; Solve the problem 2 + 5
        add a, 5
        ld h,0
        ld l, a

        B_CALL _DispHL
        B_CALL _getKey
        B_CALL  _ClrLCDFull
        ret

Number_Two:
        .db 2
```

That recalls a variable. To store a value to a variable (for right now), register A must contain the value that you want to store in the variable. Then use ld (Variable), A. On the next page is an example. It will store the answer to the addition problem 5 + 7, so that you do not lose the answer to the problem.

```
#include “ti83plus.inc”
.org  40339
.db    t2ByteTok, tAsmCmp
          B_CALL  _ClrLCDFull
          ld a, 5
; Solve the problem 5 + 7
          add a, 7
          ld (Addition_Answer), a
;Now recall and display the answer.
          ld a, (Addition_Answer)
          ld h,0
          ld l, a

          B_CALL _DispHL
          B_CALL _getKey
          B_CALL  _ClrLCDFull
          ret

Addition_Answer:
          .db 0
```

You have enough now to start your project ASM Gorillas, so in the next lesson we will begin to work on it.