TI-83+ Z80 ASM for the Absolute Beginner

LESSON FIFTEEN:

- *Storing and Retrieving Values Larger Than One Byte*
- *ASM Gorillas, Part V: The Splash Screen*

STORING AND RETRIEVING VALUES LARGER THAN ONE BYTE

As a review from Lesson 5, you can stores values to variables by using register A. You can also retrieve values from those variables by using register A.

Have you ever played Donkey Kong Country? (As a big thank you for the time I spent putting these lessons together, I'd love to see someone make Donkey Kong Country for a Ti-83+ or Ti-84+)

Donkey Kong can collect up to 100 bananas, and then he gains an extra life. Then the number of bananas is reset to zero. Thus, the

number of bananas can never exceed 255. So if someone decided to make a Donkey Kong Country game for a Z80 processor, register A would be perfect for this.

```
; Donkey Kong has collected 5 bananas


ld  a, (Number_Of_Bananas)
add a, 5
ld  (Number_Of_Bananas), a
```

For a game that requires a high score, however, one byte of numbers is not going to be enough…255 is way too much of a limit for high scores. And what about when your calculator has numbers such as 256, 341? That's WAY bigger than 255.

YET, the Z80 processor can do such things. In other words, since one-byte values aren't enough for everything, I'll show you throughout the lessons how to handle the bigger numbers. However, you need to be aware that the bigger the number is, the harder it is to work with said number. Even storing and retrieving values for two-byte numbers (0 – 65535) requires a little bit of work. This lesson, we will focus on variables holding 2-byte values.

You can only use Register A directly to store and retrieve values that are one byte in size. However, you can use HL, DE, and BC to store and retrieve values TWO bytes in size.

For this example program, I will be using hexadecimal numbers. (Remember that hexadecimal numbers are just numbers, a different "language" of numbers.) I am doing this because when you put two hexadecimal numbers together (such as 2 registers), you get the same number. If register H = $EF and register L = $23, HL = $EF23. (Recall Lesson 12.) However, with decimal numbers, this is not the case. If H = 10 and L = 234, HL does not necessarily equal 10234. For this lesson, **it is important to visualize HL as H and L put together.**

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

        ld  de, $FC23                   ; $FC23 in hexadecimal is equal to the number 64547
        ld  (Score), de
        ld  hl, (Score)

        B_CALL _DispHL
        B_CALL _getKey

        ret

Score:

 .dw  0     ; Since we are storing and retrieving the value of a 2-Byte number, we use dw instead of db.
```

What happens is instead of using one register (A) to store a one-byte value, two bytes (H and L, D and E, etc) are used to store two one-byte values. Just remember that a two-byte variable requires—of course—two bytes of RAM.

So, are you ready to call me a liar? Seems too easy, huh? It does. Well, before you toss out the lesson and decide that I don't know my right hand from my left hand, try the next example program.

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

        ld  hl, (Score)

        B_CALL _DispHL
        B_CALL _getKey

        ret

Score:

 .db $FC
 .db $23
```

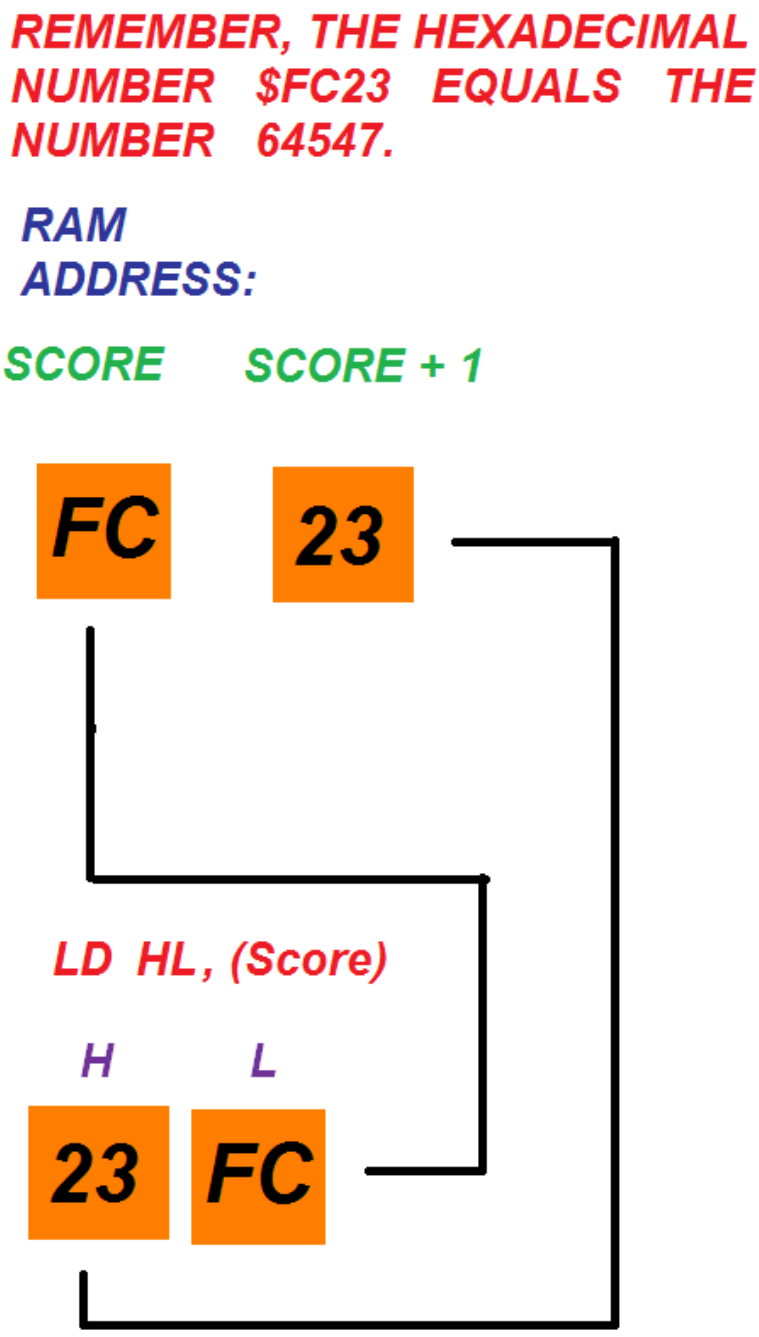
What? When _DispHL is called, do you see the value 64547? No, you should see 9212.

I suspect that the Ti-83+ processor was designed on April Fool's Day, because the designers most likely decided to play a big joke on developers.

Here's the deal: LD HL, (Score) is the same as saying the following:

LD L, (Score)

LD H, (Score + 1)



Why is it not the other way around? Why not H first, and then L? That's the joke. The people who made the processor were trying to be funny. (Okay, so I don't know the real reason. But let me have some fun ☺)

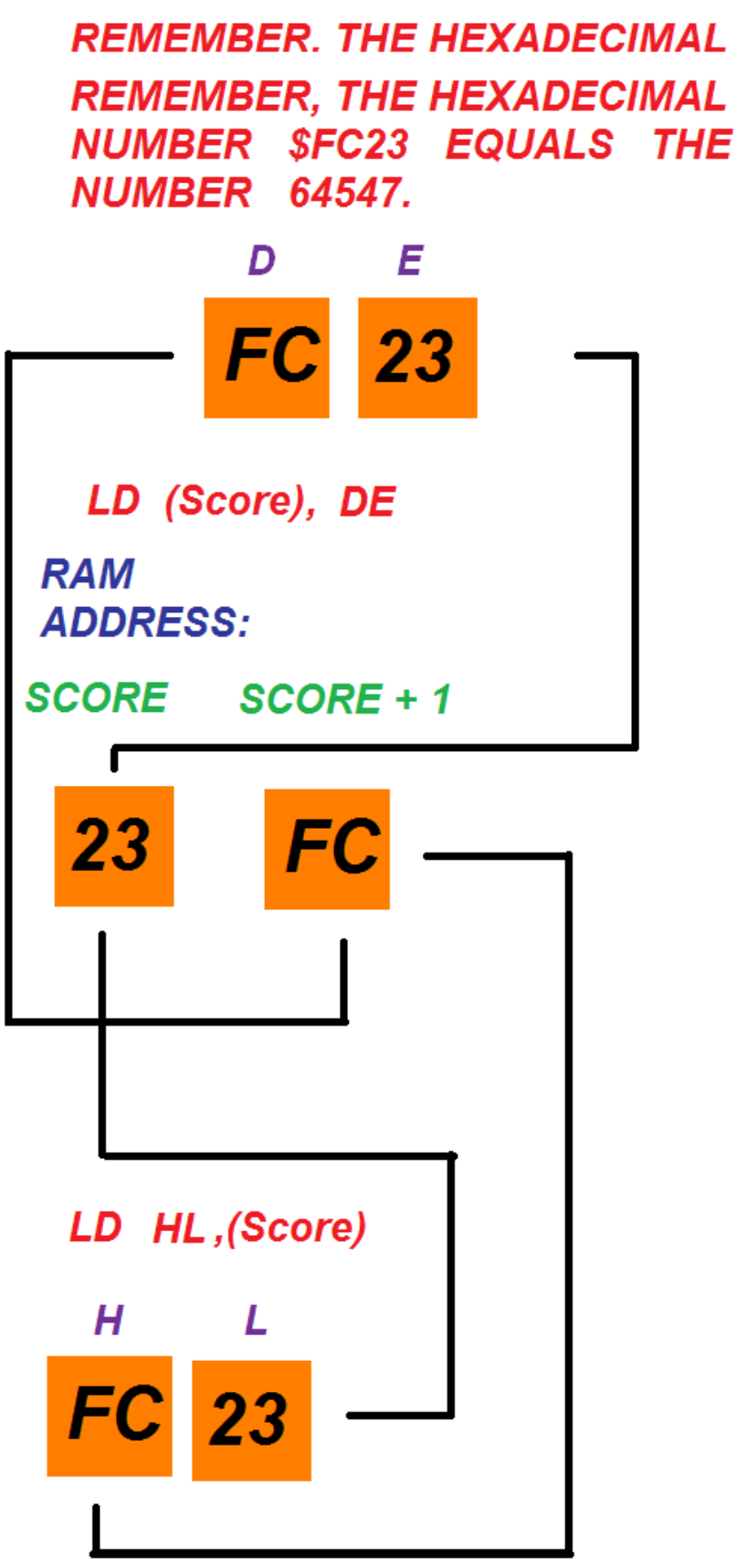
This happens also with ld DE, (Two-Byte Variable) and ld BC, (Two-Byte Variable). E is retrieved before D is, and C is retrieve before B is.

Why, then, did we not have any issues in the first example program? Because the registers are also switched with ld (Two-Byte Variable), HL/DE/BC. ld (score), HL is the same as saying

ld (score), L

ld (score + 1), H

As you will see in the diagram on the next page, (illustrating the first example program), this "reversing of values" occurs twice. So when you reverse the position of two values, and then reverse them again, you get back where you started.


REMEMBER. THE HEXADECIMAL
REMEMBER, THE HEXADECIMAL NUMBER $FC23 EQUALS THE NUMBER 64547.
D
E
FC
23
LD (Score), DE
RAM ADDRESS:
SCORE
SCORE + 1
23
FC
LD HL,(Score)
H
L
FC
23

This "reversing of values" that occurs with 2-byte variables and registers is known as **little-endian.** I'm telling you this so that when you ask for help and people bring up this term, you know what they're talking about. The important thing is, remember that if you store/retrieve values of 2-byte variables using two-byte registers, L will get stored/retrieved first, than H. E comes before D. C comes before B.

By the way, with SPASM, (be careful, this technique is for SPASM) you can replace the two .db statements with either .dw $FE23 or .dw 64547. Why did I not do this in the first place? Because I wanted you to be aware of the little-endian problem. SPASM takes the .dw statement and places the numbers in the right positions so as to help avoid the little-endian problem. But NOT ALL COMPILERS DO THIS. Be very careful when working with .dw, and make sure you know how your compiler handles little-endians.

ASM GORILLAS, PART V: THE SPLASH SCREEN

In the current state of ASM Gorillas, the menu screen looks very boring. So we are going to make it look better with a background picture!

On the next page is the picture data for this background picture. It is in a very small font, but all you need to do is copy and paste the data. At the very bottom of ASMGorillasMain.asm, create a label called Splash_Screen (with a colon at the end). Then paste the picture data immediately following it.

```
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000
.db %00001110,%00000111,%10001111,%00001110,%00000011,%11000000,%11110000,%11111100,%01101100,%00110000,%00111000,%00011110
.db %00001110,%00001100,%11001111,%00011110,%00000110,%01100001,%10011000,%11000110,%01101100,%00110000,%00111000,%00110011
.db %00011011,%00011000,%00001011,%00011100,%00001100,%00110011,%00001101,%10000110,%11011000,%01100000,%01101100,%01100000
.db %00011011,%00011000,%00011011,%00111100,%00011000,%00000110,%00001101,%10000110,%11011000,%01100000,%01101100,%01100000
.db %00110011,%00001110,%00011011,%01101100,%00011000,%00000110,%00001101,%10001100,%11011000,%01100000,%11001100,%00111000
.db %00110011,%00000111,%00011011,%01101100,%00011000,%11110110,%00001101,%11111000,%11011000,%01100000,%11001100,%00011100
.db %00110011,%00000001,%10011011,%11001100,%00011000,%00110110,%00001101,%10010000,%11011000,%01100000,%11001100,%00000110
.db %01111111,%10000001,%10110011,%10011000,%00011000,%00110110,%00011011,%00011001,%10110000,%11000001,%11111110,%00000110
.db %01100001,%10110011,%00110011,%10011000,%00001100,%01100011,%00110011,%00001101,%10110000,%11000001,%10000110,%11001100
.db %11000001,%10011110,%00110011,%00011000,%00000111,%11000001,%11100011,%00001101,%10111111,%11111111,%00000110,%01111000
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00011000
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00011000
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%01111110
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%10111101
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%10011001
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00100100
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00100100
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11111111,%10000000,%11111111
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11111111,%10000000,%10000000
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11001001,%10000000,%10000110
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11001001,%10000000,%10000110
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11111111,%10000000,%10000000
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11111111,%10000000,%10000000
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11001111,%10000000,%10000110
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11001111,%10000000,%10000110
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11111111,%10000000,%11111111,%10000000,%10000000
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11111111,%10000000,%11111111,%10000000,%10000000
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11001111,%10000000,%11001111,%10000000,%10110110
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11001111,%10000000,%11001111,%10000000,%10110110
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11111111,%10000000,%11111111,%11111111,%10000000
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11111111,%10000000,%11111111,%10000000,%10000000
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11001111,%10000000,%11001001,%10110110,%10110000
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11001111,%10000000,%11001001,%10110110,%10110000
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11111111,%11111111,%11111111,%10000000,%10000000
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11111111,%10000000,%11111111,%10000000,%10000000
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11001001,%10000110,%11001001,%10110000,%10110110
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11001001,%10000110,%11001001,%10110000,%10110110
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11111111,%10000000,%11111111,%10000000,%10000000
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11111111,%10000000,%11111111,%10000000,%10000000
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11001111,%10110000,%11111001,%10000110,%10110110
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11001111,%10110000,%11111001,%10000110,%10110110
.db %00000000,%00000000,%00000000,%00000000,%00000000,%00000000,%11111111,%11111111,%10000000,%11111111,%10000000,%10000000
.db %00000000,%00000000,%00000000,%00000000,%00011000,%00000000,%10000000,%11111111,%10000000,%11111111,%10000000,%10000000
.db %00000000,%00000000,%00000000,%00000000,%00011000,%00000000,%10110110,%11001001,%10000110,%11001111,%10110000,%10000000
.db %00000000,%00000000,%00000000,%00000000,%01111110,%00000000,%10110110,%11001001,%10000110,%11001111,%10110000,%10000000
.db %11111111,%10000000,%11111111,%10000000,%10111101,%00000000,%10000000,%11111111,%10000000,%11111111,%10000000,%10000000
.db %10000000,%10000000,%11111111,%10000000,%10011001,%00000000,%10000000,%11111111,%10000000,%11111111,%10000000,%10000000
.db %10000000,%10000000,%11001111,%10000000,%00100100,%00000000,%10000110,%11001001,%10110000,%11001001,%10110000,%10110000
.db %10000000,%10000000,%11001111,%10000000,%00100100,%00000000,%10000110,%11001001,%10110000,%11001001,%10110000,%10110000
.db %10000000,%11111111,%11111111,%11111111,%11111111,%10000000,%10000000,%11111111,%10000000,%11111111,%10000000,%10000000
.db %10000000,%10000000,%11111111,%10000000,%11111111,%10000000,%10000000,%11111111,%10000000,%11111111,%10000000,%10000000
.db %10110110,%10110110,%11111111,%10000110,%11001111,%10000000,%10110110,%11111001,%10000110,%11001001,%10110110,%10110110
.db %10110110,%10110110,%11111111,%10000110,%11001111,%10000000,%10110110,%11111001,%10000110,%11001001,%10110110,%10110110
.db %10000000,%10000000,%11111111,%10000000,%11111111,%11111111,%10000000,%11111111,%10000000,%11111111,%10000000,%10000000
.db %10000000,%10000000,%11111111,%10000000,%11111111,%10000000,%10000000,%11111111,%10000000,%11111111,%10000000,%10000000
.db %10110110,%10110000,%11001111,%10110110,%11001111,%10110000,%10110000,%11001001,%10000110,%11001111,%10110110,%10110000
.db %10110110,%10110000,%11001111,%10110110,%11001111,%10110000,%10110000,%11001001,%10000110,%11001111,%10110110,%10110000
.db %10000000,%10000000,%11111111,%10000000,%11111111,%10000000,%10000000,%11111111,%10000000,%11111111,%10000000,%10000000
```

Normally, we would use B_CALL _GrBufCpy to display an image stored in plotsscreen. However, as always, B_CALL routines are slow. As you will see in this lesson, we will have to display the splash screen many, many times. But _GrBufCpy causes the screen to blink because of how slow it is. And that will cause people's eyes to hurt. So we need another function to do what _GrBufCpy does, but faster.

On this page, I have attached some code for you to paste after the picture data. DO NOT TRY TO UNDERSTAND THE CODE. It has some stuff you have not learned yet.

```
fastCopy:

        ld a,$80
        out ($10),a
        ld hl,plotsscreen-12-(-(12*64)+1)
        ld a,$20
        ld c,a
        inc hl
        dec hl
_
        ld b,64
        inc c
        ld de,-(12*64)+1
        out ($10),a
        add hl,de
        ld de,10
_
        add hl,de
        inc hl
        inc hl
        inc de
        ld a,(hl)
        out ($11),a
        dec de
        djnz -_
        ld a,c
        cp $2B+1
        jr nz,--_


        ret
```

Now, open ASMGorillasStartProgram.asm. Type the following after Display_New_Menu:

```
ld hl, Splash_Screen
ld de, plotsscreen
ld bc, 768
ldir
```

We copy the background picture to plotsscreen. We won't see the picture yet, but the data will be there, waiting to be displayed.

Now, open the file ASMGorillasMenus.asm. Look for the following lines of code:

```
Display_AI_Difficulty_Menu:

    ld hl, AI_Difficulty_Menu_Text
    ld b, AI_Difficulty_Menu_Items

    ld a, AI_Difficulty_Menu_End_Of_Menu
    ld (End_Of_Menu), a

    jr Continue_To_Display_Menu


Continue_To_Display_Menu:

    ld a, c
    push af
```

Immediately after "push af," type in the following:

```
SET  textWrite, (IY + sGrFlags)
```

I don't expect you to know what this means, at least for right now. However, I will tell you why we include this line. Normally, when text is displayed, it shows up on the screen immediately. But we don't want to do that yet, not in this case. We want to put the text in plotsscreen, so that when we are ready to display the background—and its menu—we can everything at once. The line of code you just typed in tells the calculator to do just that—store the text in plotsscreen and not display it until we want it to.

After the line call Display_Text_Menu, type in the following:

```
call  fastcopy
```

This is what we use instead of B_CALL _GrBufCpy. Fastcopy takes whatever is inside of plotsscreen, and displays it immediately.

Now go to the following part in the file:

```
;Display the cursor while saving which menu we are displaying

    ld a, 14
    ld (Menu_Y), a

Move_Cursor_Loop:
```

Now, type the following after Move_Cursor_Loop:

RES textWrite, (IY + sGrFlags)

This will draw the cursor immediately to the screen, not to plotsscreen. We want the player to see his cursor immediately.

Now compile your program, and you should see something like this rather than a plain old background!