TI-83+ Z80 ASM for the Absolute Beginner

LESSON THIRTEEN:

- *Of Mice and Men: The Sequel*
- *ASM Gorillas, Part IV: Using Keys to Navigate Through the Menus*

OF MICE AND MEN: THE SEQUEL

This time, we will not be speaking "Of Registers and RAM." We will be speaking "Of Calls and the Stack." Remember that the stack is what holds values that are "pushed," saving the values of registers that we want to save. We "pop" a value from the stack to recall it.

The Ti-83+ processor has a special register called PC, a register which is used only by the processor (you absolutely cannot use it in your program). Do you remember from lesson 4-5 that your ASM program runs in RAM, starting at $9D93? And that the more the program runs, the further in RAM you go? This value, the place in RAM containing the code being executed at the time, is kept track of in register PC. So when your program starts "officially" at RAM address $9D95, PC will be $9D95. After 100 instructions each 1 byte long, PC will be equal to $9DF9. ($9D95 + 100 equals $9DF9)

PC doesn't just hold the value of where the program is running, but **the program will run exactly what PC tells it to run**. If you were somehow able to change PC to be equal to $E470, the calculator would run whatever was stored at $E470.

What's the big deal? Well, when you use the instruction CALL Label_Routine, then exit the function Label_Routine with the instruction RET, the program needs to know where to return. As you might expect, it should return to where the program initiated the call. So if you, hypothetically, have CALL Play_Game at $9E46, PC will at that particular time equal $9E46. Then when you RET from Play_Game to return to the main program, you should return to $9E49. (This is because you want to return to whatever is AFTER the CALL instruction, and the CALL instruction takes 3 bytes. $9E46 + 3 = $9E49.)

But the calculator needs to remember that PC will equal $9E49 when it returns! Otherwise, when the calculator reaches the instruction "RET", it will jump to some random place because it doesn't remember where to return! So when you use CALL, the value in register PC is **pushed onto the stack**, where you store values of other 2-Byte registers. Then the location of your routine (such as Play_Game) is, afterwards, stored in PC and your calculator will run the routine. Then when you RET, that value is popped into PC so your calculator can run from that point.

The reason this is so important to remember is CALL uses exactly the same stack that you use PUSH and POP for when it comes to AF, HL, BC and DE. It is very important that you don't accidentally pop the value that the program was supposed to return to. Otherwise, when you use a RET instruction, a random value will be popped into PC, and your calculator will jump to some random address.

Let's say you have the following program:

```
LD BC,  453
PUSH BC
CALL Store_Value_Into_HL        ; Let's suppose that this instruction is at the RAM address $9DB1
                                ;Then the value $9DB4 will be pushed, since that is where the calculator
                                ; needs to return afer finishing "Store_Value_Into_HL.
```

So our stack looks like this:

$9DB4 (Hexadecimal at the top of the stack)

453 (Decimal Number)

So say we immediately want to retrieve our saved value from BC. We call POP BC, but does BC = 453? No! It equals $9DB4, since this value was at the top of the stack! Furthermore, when the calculator reaches the instruction "RET", PC will equal 453! So the calculator will jump to 453, causing random results in your program.

I've made that mistake countless times. When you write an ASM program, always make sure that you don't accidentally POP the value of the place your calculator should return to. Otherwise, your program will act very strangely, jumping to some random area. And never forget that RET always pops the top value off the stack

ASM GORILLAS, PART IV: USING KEYS TO NAVIGATE THROUGH THE MENUS

Last time we worked with ASM Gorillas, we could only display a certain menu, and we couldn't switch from menu to menu. We are going to change that in this lesson. Once again, there are changes I had to make since the last ASM Gorillas lesson, besides simply adding code. As good practice, we will simply make these changes in the lesson.

It's quite obvious that just like a menu has a beginning, it has an end. Even the extremely long Catalog menu has an end, a "last item" if you will. (This last item is a question mark.) Now what happens when your cursor reaches the question mark? What if you press down again? You've reached the end of the menu, so you can't scroll down anymore. Instead, the cursor moves to the top of the menu again.

This idea of the menus is the approach we'll be taking for ASM Gorilla Menus. When the user makes it to the top of the menu and presses the up key again, the cursor will move to the bottom of the menu. Furthermore, when the user makes it to the bottom of the menu and presses the down key again, the cursor will go to the top of the menu.

The thing to consider is, not all menus have the same number of items—some have four items, some have three, and some have five. So sometimes the last item in the menu will be the fourth one, sometimes the last item in a menu will be the third item, and sometimes it will be item number 5. We are going to tell the calculator where the end of the

menu is, depending on the menu selected. However, we're not going to say "The last item is the fourth item" or "The last item is the fifth item." Instead, we're going to say the following: "When the cursor is pointing to the last menu item, this is how many pixels down it will be…"

Recall that the first line of text of the menu is displayed at Y = 14. Also recall that each line of the menu takes up seven pixels. So if a menu has 4 items, and IF the cursor is at the last item, the cursor will be at Y = 14 + 7 +7 + 7 = 35. If a menu has 5 items, and IF the cursor is at the last item, the cursor will be at Y = 14 + 7 + 7 + 7 + 7 = 42.

Go to ASMGorillasConstants.asm, and place your cursor directly below the line "AI_Difficulty_Menu_Items .equ 5". Type in the following (lines in bold have already been typed, so that you know exactly where to type):

```
Main_Menu_Items .equ 4
Settings_Menu_Items .equ 4
Players_Menu_Items .equ 3
AI_Or_Human_Menu_Items .equ 3
AI_Difficulty_Menu_Items .equ 5

Main_Menu_End_Of_Menu .equ 35
Settings_Menu_End_Of_Menu .equ 35
Players_Menu_End_Of_Menu .equ 28
AI_Or_Human_Menu_End_Of_Menu .equ 28
AI_Difficulty_Menu_End_Of_Menu .equ 42

Player1NameX .equ 0
Player1NameY .equ 0
Player2NameX .equ 51
```

We just added five new constants. We will use these 5 constants very soon. For now, we're going to create a new file.

Create a new file entitled "ASMGorillasStartProgram.asm" As you can imagine, this will be code that is run ONCE at a time, rather than over and over. Consider it "setup" code.

```
        B_CALL _ClrLCDFull          ;Clears the screen

        B_CALL _RunIndicOff         ;Turns off the little line you usually see running at the upper-right hand corner
                                    ;of the screen
Display_New_Menu:

        ld d, MainMenuItem1X
        ld e, MainMenuItem1Y
        ld c, a                     ;Saves the value of our menu for later
        call Menu_Before_Game
        jr Display_New_Menu

Quit_Program:

        B_CALL _ClrLCDFull
        ret
```

Open "ASMGorillasMain.asm", and add the following line after #include "ti83plus.inc"

#include "ASMGorillasStartProgram.asm"

Add the following into your list of variables in this file:

```
End_Of_Menu:
    .db 0
```

Finally, go to "ASMGorillasMenus.asm" and replace the top four lines with the following line:

Menu_Before_Game:

Menu_Before_Game is a label that the program will jump to when it needs to display a different menu. For instance, if the user needs to go from the Main Menu to the Settings Menu, the program will jump to Menu_Before_Game.

Now, in ASMGorillasMenus.asm, scroll down the file to Display_Main_Menu. Once again, the lines below that are not in bold should be added between the bold lines.

```
Display_Main_Menu:
        push de                 ; Saves our menu position, since _ClrLCDFull will destroy whatever is
                                ; inside of de
        B_CALL _ClrLCDFull      ;This subroutine does NOT clear register C, which is why
                                ;we saved register A into register C
        pop de
        ld hl, Main_Menu_Text
        ld b, Main_Menu_Items
        ld a, Main_Menu_End_Of_Menu
        ld (End_Of_Menu), a

        jr Continue_To_Display_Menu

Display_Settings_Menu:

        ld hl, Settings_Menu_Text
        ld b, Settings_Menu_Items
        ld a, Settings_Menu_End_Of_Menu
        ld (End_Of_Menu), a             ;CODE CONTINUED ON NEXT PAGE

        jr Continue_To_Display_Menu
```

```
Display_AI_Or_Human_Menu:

        ld hl, AI_Or_Human_Menu_Text
        ld b, AI_Or_Human_Menu_Items
        ld a, AI_Or_Human_Menu_End_Of_Menu
        ld (End_Of_Menu), a


        jr Continue_To_Display_Menu

Display_Players_Menu:

        ld hl, Players_Menu_Text
        ld b, Players_Menu_Items
        ld a, Players_Menu_End_Of_Menu
        ld (End_Of_Menu), a


        jr Continue_To_Display_Menu

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
```

Do you see what we just did? We just told the calculator where the end of each menu will be. The calculator will now know at which point the cursor should no longer scroll down.

Now go to Continue_To_Display_Menu, and remove the 4 lines immediately following that label. Then start typing after Continue_To_Display_Menu. This is where we will add the code to tell the calculator how to handle keypresses in a menu setting. Incidentally, I will provide all of "ASMGorillasMenu.asm" at the end of this lesson so that you can make sure you have everything, in case I missed something. (I always test my programs before I send out a lesson)

```
ld a, c
push af
```

Remember that register A must contain the menu we need to display. So we load it from register C. Whatever is stored in Register C will be erased when the program runs, so we save register A by pushing af.

```
call Display_Text_Menu
```

Recall that this is the sub that actually draws the menu, depending on what menu we need to display. So we call this sub so that the menu will be drawn on the screen.

```
    ld a, 14
    ld (Menu_Y), a
```

Since we use Menu_X and Menu_Y for holding the position of the cursor, and since the cursor starts at the top (Y = 14), we set the Y cursor position to 14.

```
Move_Cursor_Loop:
    call Allow_User_To_Move_Cursor
    jr Move_Cursor_Loop
```

Move_Cursor_Loop is a loop that occurs every time a key is pressed. This way, we can get one key press after another after another. In other words, the user presses a key and moves the menu cursor, and then is allowed to press the key again to move the cursor again.

Remember, remember, REMEMBER, that when we run "CALL Allow_User_To_Move_Cursor," the calculator needs to know where to return. So the value telling the calculator where to return is pushed onto the stack. Then when a RET is encountered, this value can be popped and the calculator will know exactly to return. This is important for our code later in this lesson.

There is nothing else to type before the label Display_Text_Menu. Go to the **ret** statement at this label. It's the ret after djnz Display_Text_Menu.

Now we're going to add the sub Allow_User_To_Move_Cursor. This is responsible for drawing the cursor, and allowing the cursor to move it up and down.

```
Allow_User_To_Move_Cursor:

        ld a, 2
        ld (penCol), a
        ld a, (Menu_Y)
        ld (penRow), a
```

We set the coordinates for the cursor. The cursor will always be displayed at X = 2, but of course, the Y position will change as the player moves the cursor up and down.

```
        ld a, $05 ; The code for the right arrow key
        B_CALL _VPutMap
```

B_CALL _VPutMap will place a single character, not a string of text, on the screen. The character we want to display is stored in register

A. As another example, if you want to display the character “A” and **only** that character, do the following: ld a, $41 B_CALL _VPutMap

```
B_CALL _getKey
cp kDown
jr z, Move_Menu_Cursor_Down
cp kUp
jr z, Move_Menu_Cursor_Up
cp kEnter
jr z, Menu_Select_Item
jr Allow_User_To_Move_Cursor
```

Menu_Select_Item will be a function we add later, selecting what a user does depending on what item he has selected when the Enter key has been pressed. Also, if the user presses an illegal key, the function will restart and wait for another key press.

```
Move_Menu_Cursor_Down:


    ld a, 2
    ld (penCol), a
    ld a, (Menu_Y)
    ld (penRow), a


    ld a, $06 ; The code for the right arrow key
    B_CALL _VPutMap
```

Character $06 is just a blank space. We use this to clear where our previous cursor was drawn.

Our next step after the “down” key has been pressed is to see if the cursor will go too far down the menu. If so, we want to move the cursor to the top of the menu.

```
Move_Menu_Cursor_Down:


    ld a, (End_Of_Menu)
    ld e, a
    ld a, (Menu_Y)


    cp e
    jr nc, Reset_Move_Menu_Cursor_Down
```

We retrieve how many pixels down the end of the menu is. Storing this value into register E (to save it), we then look at how many pixels down the cursor is. CP E means, "Let's see where the cursor is, and compare it to where the end of the menu is." If it turns out that the cursor is as the end of the menu, **we don't want it to go down any further!** Instead, we move it to the top of the menu.

```
    add a, 7
    ld (Menu_Y), a
    ret
```

If the program DOES NOT jump to Reset_Move_Menu_Cursor_Down, that means the cursor is not at the end of the menu yet. Thus, we can safely advance the cursor 7 pixels down. Then we leave the sub to detect another keypress.

```
Reset_Move_Menu_Cursor_Down:

    ld a, 14
    ld (Menu_Y), a
    ret
```

Recall that the top of the menu is located 14 pixels down. We reset the cursor to that position if the cursor goes too far down the menu.

```
Move_Menu_Cursor_Up:

    ld a, 2
    ld (penCol), a
    ld a, (Menu_Y)
    ld (penRow), a

    ld a, $06 ; The code for the right arrow key
    B_CALL _VPutMap
```

```
    ld a, (Menu_Y)
    sub 7

    cp 7
    jr z, Reset_Move_Menu_Cursor_Up

    ld (Menu_Y), a ;If we aren't at the top
                   ;of the menu and can safely
                   ;move up, we already
                   ;subtracted 7 pixels and
                   ;don't need to do so again
    ret

Reset_Move_Menu_Cursor_Up:

    ld a, (End_Of_Menu)
    ld (Menu_Y), a ;Resets the cursor
                   ;to the bottom of the menu
    ret
```

The only thing I need to explain is the underlined lines. Supposing that, hypothetically, the cursor is at the top of the menu, Menu_Y will equal 14. Thus, if "up" is pressed at Y = 14, the cursor will go too far up, and we want to prevent that.

Of course, just like we move the cursor 7 pixels down when the down key is pressed, we move the cursor up 7 pixels when the up key is pressed. SO, if the cursor is at Y = 14, then moving the cursor to Y = 14 – 7 (in other words, Y = 7) will bring the cursor past the top of the menu. We subtract 7 from register A = Menu_Y to see if we will indeed be at Y = 7, past the top of the menu.

Now, what should happen when the enter key is pressed? Well, there many possible things that could happen, depending on the item select. For this lesson, we'll focus on switching from one menu to another.

Now, go to the bottom of the file and continue typing this new text. I'm afraid I won't be explaining much, because the text is pretty redundant. However, I have a couple of new things to teach, so I will explain the lines in bold.

```
Menu_Select_Item:


        ld a, (Menu_Y)
        cp 14                   ; If the cursor is at Y = 14, it is pointing to the first item in the menu.
        jr z, First_Item_Selected
        cp 21
        jr z, Second_Item_Selected
        cp 28
        jr z, Third_Item_Selected
        cp 35
        jr z, Fourth_Item_Selected
        cp 42
        jr z, Fifth_Item_Selected


First_Item_Selected:


        pop af          ; Since every menu has a different first item, we need to know which menu
        pop af          ; the player is currently on.  HOWEVER, the first value we popped is NOT the menu the player
                        ; is using.  It is the value that was pushed when the program ran
                        ; CALL Allow_User_To_Move_Cursor.  We clear this value, since we DO NOT need to return
                        ; to where the call was initiated.  Then we pop af again so that register A now holds the menu
                        ; the player currently is on.
```

```
        cp Main_Menu                ; Depending on the menu (and register A tells us what the menu is),
                                    ; either quit the game or go to a different menu
        jr z, Goto_Ai_Or_Human_Menu

        cp AI_Or_Human_Menu
        jr z, Goto_AI_Difficulty_Menu

        cp Ai_Difficulty_Menu
        jr z, Goto_Players_Menu

        ret

Second_Item_Selected:

        pop af
        pop af

        cp Ai_Difficulty_Menu
        jr z, Goto_Players_Menu

        cp AI_Or_Human_Menu
        jr z, Goto_Players_Menu

        ret
```

```
Third_Item_Selected:
        pop af
        pop af

        cp Players_Menu
        jr z, Goto_AI_Or_Human_Menu

        cp AI_Or_Human_Menu
        jr z, Goto_Main_Menu

        cp Ai_Difficulty_Menu
        jr z, Goto_Players_Menu

        cp Main_Menu
        jr z, Goto_Settings_Menu
        ret
Fourth_Item_Selected:

        pop af
        pop af

        cp Ai_Difficulty_Menu
        jr z, Goto_Players_Menu

        cp Settings_Menu
        jr z, Goto_Main_Menu

        cp Main_Menu
        jr z, Quit_Game_From_Main_Menu
        ret
```

```
Fifth_Item_Selected:
        pop af
        pop af

        cp Ai_Difficulty_Menu
        jr z, Goto_AI_Or_Human_Menu
        ret

Goto_Ai_Or_Human_Menu:          ; We specify a new menu. Then RET will return us to
                                ; Display_New_Menu.  This is because this is the next value in the stack,
                                ; telling the calculator where to return.
        ld a, AI_Or_Human_Menu
        ret

Goto_AI_Difficulty_Menu:
        ld a, AI_Difficulty_Menu
        ret

Goto_Players_Menu:
        ld a, Players_Menu
        ret

Goto_Main_Menu:
        ld a, Main_Menu
        ret
Goto_Settings_Menu:
        ld a, Settings_Menu
        ret
```

So hopefully, a lot of this is relatively easier to understand. However, we now need to tell the calculator what to do when the user selects "QUIT." To do this, I'm going to introduce a rather interesting technique.

Our stack now holds two important values. The top of the stack holds the value telling the calculator where to return—that is, somewhere in Display_New_Menu. (The second value tells the calculator where to go to return to the operating system.) However, we don't want to go to Display_New_Menu. We want to quit the game! We want to go to the label Quit_Program.

Why don't we just JP or JR to that label? Because there are still values in the stack, and we want the first one cleared, or else the calculator will likely crash.

We could just pop this value and then jump to Display_New_Menu. However, I'm going to teach you another way. Remember how RET pops a value that tells the calculator where to go? We're going to force the calculator to go to Quit_Program after a RET is encountered.

The calculator has a register called SP. It hold where the top of the stack is. Whenever we push a value onto the stack, or when we pop a value, SP is adjusted appropriately, since the stack grows and shrinks. We can store a value into the stack by using register HL. So we will store the location of Quit_Program into the top of the stack. Then when RET is encountered, this value will be popped, and the calculator will thus go to Quit_Program!

Quit_Game_From_Main_Menu:

```
pop hl
ld hl, Quit_Program
push hl
ret
```

At this point, our lesson is done. Feel free to check the next few pages, to make sure you have all of ASMGorillasMain.asm typed correctly.

```
;hl is the location of the menu text
;b is the number of items the menu has
;d contains the X position, the column to display the text at.
;e contains the Y position, the row to display the text at.

Menu_Before_Game:

     push de

     B_CALL _ClrLCDFull

     pop de


     cp Main_Menu
     jr z, Display_Main_Menu
     cp Settings_Menu
     jr z, Display_Settings_Menu
     cp AI_Or_Human_Menu
     jr z, Display_AI_Or_Human_Menu
     cp Players_Menu
     jr z, Display_Players_Menu
     cp AI_Difficulty_Menu
     jr z, Display_AI_Difficulty_Menu



Display_Main_Menu:

     ld hl, Main_Menu_Text
     ld b, Main_Menu_Items
     ld a, Main_Menu_End_Of_Menu
     ld (End_Of_Menu), a

     jr Continue_To_Display_Menu

Display_Settings_Menu:

     ld hl, Settings_Menu_Text
     ld b, Settings_Menu_Items
     ld a, Settings_Menu_End_Of_Menu
     ld (End_Of_Menu), a


     jr Continue_To_Display_Menu

Display_AI_Or_Human_Menu:

     ld hl, AI_Or_Human_Menu_Text
     ld b, AI_Or_Human_Menu_Items
     ld a, AI_Or_Human_Menu_End_Of_Menu
     ld (End_Of_Menu), a
```

```
     jr Continue_To_Display_Menu

Display_Players_Menu:

     ld hl, Players_Menu_Text
     ld b, Players_Menu_Items
     ld a, Players_Menu_End_Of_Menu
     ld (End_Of_Menu), a


     jr Continue_To_Display_Menu

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



     call Display_Text_Menu

     ;Display the cursor while saving which menu we are
displaying

     ld a, 14
     ld (Menu_Y), a

Move_Cursor_Loop:


     call Allow_User_To_Move_Cursor


     jr Move_Cursor_Loop


Display_Text_Menu:



     ld a, e
     ld (penRow),a
     ld a, d
```

```
      ld (penCol), a



      ld c, b         ;Save the number of menu items
                      ;left to display
      ld b, (hl)
      inc hl

      B_CALL _VPutSN

      ld a, e         ;Move the text line of text to the next
                      ;row
      add a, 7
      ld e, a

      ld b, c
      djnz Display_Text_Menu


      ret

Allow_User_To_Move_Cursor:

      ld a, 2
      ld (penCol), a
      ld a, (Menu_Y)
      ld (penRow), a

      ld a, $05 ; The code for the right arrow key
      B_CALL _VPutMap


      B_CALL _getKey
      cp kDown
      jr z, Move_Menu_Cursor_Down
      cp kUp
      jr z, Move_Menu_Cursor_Up
      cp kEnter
      jr z, Menu_Select_Item
      jr Allow_User_To_Move_Cursor

Move_Menu_Cursor_Down:

      ld a, 2
      ld (penCol), a
      ld a, (Menu_Y)
      ld (penRow), a

      ld a, $06 ; The code for the right arrow key
      B_CALL _VPutMap


      ld a, (End_Of_Menu)
```

```
      ld e, a
      ld a, (Menu_Y)
      cp e
      jr nc, Reset_Move_Menu_Cursor_Down

      add a, 7
      ld (Menu_Y), a
      ret

Reset_Move_Menu_Cursor_Down:

      ld a, 14
      ld (Menu_Y), a
      ret

Move_Menu_Cursor_Up:

      ld a, 2
      ld (penCol), a
      ld a, (Menu_Y)
      ld (penRow), a

      ld a, $06 ; The code for the right arrow key
      B_CALL _VPutMap


      ld a, (Menu_Y)
      sub 7

      cp 7
      jr z, Reset_Move_Menu_Cursor_Up

      ld (Menu_Y), a
      ret

Reset_Move_Menu_Cursor_Up:

      ld a, (End_Of_Menu)
      ld (Menu_Y), a
      ret

Menu_Select_Item:

      ld a, (Menu_Y)

      cp 14
      jr z, First_Item_Selected
      cp 21
      jr z, Second_Item_Selected
      cp 28
      jr z, Third_Item_Selected
      cp 35
      jr z, Fourth_Item_Selected
      cp 42
```

```
     jr z, Fifth_Item_Selected

First_Item_Selected:

     pop af
     pop af

     cp Main_Menu
     jr z, Goto_Ai_Or_Human_Menu

     cp AI_Or_Human_Menu
     jr z, Goto_AI_Difficulty_Menu

     cp Ai_Difficulty_Menu
     jr z, Goto_Players_Menu

     ret

Second_Item_Selected:

     pop af
     pop af

     cp Ai_Difficulty_Menu
     jr z, Goto_Players_Menu

     cp AI_Or_Human_Menu
     jr z, Goto_Players_Menu

     ret

Third_Item_Selected:

     pop af
     pop af

     cp Players_Menu
     jr z, Goto_AI_Or_Human_Menu

     cp AI_Or_Human_Menu
     jr z, Goto_Main_Menu

     cp Ai_Difficulty_Menu
     jr z, Goto_Players_Menu

     cp Main_Menu
     jr z, Goto_Settings_Menu

     ret

Fourth_Item_Selected:

     pop af
     pop af
```

```
        cp Ai_Difficulty_Menu
        jr z, Goto_Players_Menu
        cp Settings_Menu
        jr z, Goto_Main_Menu
        cp Main_Menu
        jr z, Quit_Game_From_Main_Menu

        ret

Fifth_Item_Selected:


        pop af
        pop af

        cp Ai_Difficulty_Menu
        jr z, Goto_AI_Or_Human_Menu

        ret


Goto_Ai_Or_Human_Menu:

        ld a, AI_Or_Human_Menu
        ret

Goto_AI_Difficulty_Menu:

        ld a, AI_Difficulty_Menu
        ret

Goto_Players_Menu:

        ld a, Players_Menu
        ret

Goto_Main_Menu:

        ld a, Main_Menu
        ret

Goto_Settings_Menu:

        ld a, Settings_Menu
        ret

Quit_Game_From_Main_Menu:

        pop hl
        ld hl, Quit_Program
        push hl
        ret
```