TI-83+ Z80 ASM for the Absolute Beginner

APPENDIX E:

- *Miscellaneous Tips and Tricks*

MISCELLANEOUS TIPS AND TRICKS

This appendix contains some tips and tricks that you probably won't find in a typical ASM lesson. They are listed from easiest to hardest, so don't just hop back and forth between sections!

Optimized Compare and Zero Instructions

- XOR A is smaller and faster than ld a, 0
- OR A is smaller and faster than cp 0
- If you want to see if BC, DE or HL equal zero, you can put one register in a pair into A and OR it with the other register in the pair. For example:

```
LD A, D
OR E
JR Z, Register_DE_Equals_Zero
```

- CP only works for register A. However, you can use the following code to make a CP instruction that works with HL, useful for two-byte values:

```
LD DE, 1000     ; 1000, or whatever value you want to
                ; compare with HL
OR A
SBC HL, DE
ADD HL, DE
```

Local Labels

Sometimes you will have a section of code where you need many, many labels. This is usually the result of several jr and cp statements.

```
    cp 1
    jr z, Number1
    cp 2
    call z, Number2
    cp 3
    jr nc, Number3
Number1:
    add a, e
    ret
Number2:
    sub a, 5
    ret
Number3:
    cp 4
    jr z, Number1
    ret
```

For some people, coming up with a bunch of label names can, at times, be tedious or time-consuming. As a programmer, perhaps you don't mind making a bunch of label names. However, as an alternative, you can use underscores to create "local labels."

```
    cp 1
    jr z, +_
    cp 2
    call z, ++_
    cp 3
    jr nc, +++_
_
    add a, e
    ret
_
    sub a, 5
    ret
_
    cp 4
    jr z, -_
    ret
```

As you can see from this example, the number of plus signs tells the calculator to go forward to the nearest, second nearest, third nearest, etc. code with an underscore before it. Minus signs tell the calculator to go backwards.

You can have as many local labels as you want, but no more than 5 plus signs or minus signs put together. (++++++++_ is illegal) Also, +_ and _ mean the same thing. So jr z, +_ is the same thing as jr z, _.

Jumping to a location stored inside a register

If you have a program or RAM location stored inside of HL, IX or IY, you can jump to that location. Instead of

```
	jp Decrease_Number_Of_Lives
```

You can use

```
	ld HL, Decrease_Number_Of_Lives
	jp (hl)
```

This is best used when you need to jump to a location that could be anywhere, for example, when you have to jump to a label that changes location in RAM from time to time.

JP (IX) and jp (IY) work the same way.

CPI and CPIR

CPI is a useful instruction for comparing large amounts of data, especially strings stored in RAM. CPI is like CP A, (HL), except that it does not affect the carry flag, and HL is increased every time CPI is used. (This way HL will move to the next byte of data to look at)

Suppose you've programmed a game that requires passwords. The password for the first level is LITTLELAMB. (Example is on next page)

```
LD HL, Password_Entered_By_The_Player                          ; HL is the start of the data that you want to compare

LD A, 'L'                  ; Putting a letter in single quotes is perfectly acceptable.

CPI

JR NZ, Password_Is_Incorrect

;HL now equals Password_Entered_By_The_Player + 1.  CPI does this automatically.

LD A, 'I'

CPI

JR NZ, Password_Is_Incorrect

LD A, 'T'

CPI

JR NZ, Password_Is_Incorrect

LD A, 'T'

CPI

JR NZ, Password_Is_Incorrect

LD A, 'L'

CPI

JR NZ, Password_Is_Incorrect

LD A, 'E'

CPI

JR NZ, Password_Is_Incorrect

LD A, 'L'

CPI

JR NZ, Password_Is_Incorrect

LD A, 'A'

CPI

JR NZ, Password_Is_Incorrect

LD A, 'M'

CPI

JR NZ, Password_Is_Incorrect

LD A, 'B'

CPI

JR NZ, Password_Is_Incorrect


PASSWORD_IS_CORRECT:
```

Can you guess how to make this code even shorter? (Hint: LD A, (DE) is a valid instruction. So is LD A, (BC))

```
LD HL, Password_Entered_By_The_Player                ; HL is the start of the data that you want to compare

LD DE, Password_For_Level_One

LD B, 10          ;There are ten characters to check.


Check_One_Character_Of_Password:


LD A, (DE)

CPI

JR NZ, Password_Is_Incorrect

INC DE

DJNZ Check_One_Character_Of_Password      ;Even if one character of the password is correct, we have to check the rest


YES_Password_IS_Correct




Password_For_Level_One:

.db  "LITTLELAMB"
```

While CPI is great for comparing large amounts of data, CPIR is used to search RAM until a single value of your choice is found. (This is very useful, for example, to find values inside of strings and where these values occur) When you want to search for a value, put your value into register A. BC should contain how many bytes to look at before giving up. (If you want to search and search and search without giving up, BC should equal 0.) When a value is found before the “give up” limit, the Z flag is set.

InStr(is a Microsoft Basic command used to search a string to see if a smaller string of your choice can be found inside of it. Here's a Z80 version to see if the string "OTION" appears in the string "DO THE LOCOMOTION". It makes great use of both CPI and CPIR.

```
LD HL, DO_THE_LOCOMOTION_STRING

LD BC, 25                ; If "OTION" is not found within 25 characters, give up.


Search_For_Character_O:

 ld a, 'O'

 cpir         ;Search DO_THE_LOCOMOTION_STRING until the letter "O" is found

 jp po, String_Not_Found                    ; PO means BC = 0, so give up.  PO CANNOT be used with jr.

; If PO is false, the letter O was found.  HL now points to the letter after the letter O.

 ld a, 'T'

 cpi

 jr nz, Search_For_Character_O            ; If the letter after O is not "T", then obviously the string "OTION" does not exist

                                          ; at the particular location

 ld a, 'I'

 cpi

 jr nz, Search_For_Character_O

 ld a, 'O'

 cpi

 jr nz, Search_For_Character_O

 ld a, 'N'

 cpi

 jr z, String_Has_Been_Found

 jr Search_For_Character_O


DO_THE_LOCOMOTION_STRING:


 .db "DO THE LOCOMOTION"
```

Accidental 1-byte registers

As you know, register HL can be split into registers H and L. BC can be split into B and C. DE can be split into D and E.

What about IX and IY? You can split them, but that is not what the designers of the Z80 processor in your calculator intended. The ability to split these 2-byte registers into separate 1-byte registers came about as a total accident. This is an advantage and a disadvantage: You have 4 more 1-byte registers to play with (2 from splitting IX and 2 from splitting IY), but if you decide you want to do so, your game won't work on the Ti-Nspire's 84+ mode.

So, HL is split into H and L. BC is split into B and C. **IX** is split into IXH and IXL. IY is split into IYH and IYL. (Remember to use IY with extreme caution, because the operating system uses it!)

Almost anything you can do with H and L, you can do with IXH, IXL, IYH and IYL.

```
LD B, IXH

LD IXL, 34

OR IYH

ADD A, IYL

INC IXL
```

But not everything can be done with these 1-byte registers. Spasm will tell you if there's a problem.

These four 1-byte registers can be helpful, but they are slower than using other 1-byte registers. So even if you don't care about lack of Nspire compatability, you should still save these registers for instances such as when all other 1-byte registers are tied up.

Using Z, NZ, C and NC to your advantage

Did you know that you can force the Z and C flags to set or reset themselves? As you know, the Z flag is set if a calculation results in an answer of "zero" and it is reset otherwise. Similarly, the carry flag is set if there's a register overflow, otherwise it's reset.

But sometimes you might want to use these flags to represent ***some other condition.*** For example, in a program I wrote, I wanted to know if the user entered a 4-digit negative number. Even though the C flag was not created to detect negative or positive numbers, I used it for such a purpose. I forced the C flag to set itself if the user entered a negative number, and I forced the C flag to reset itself if the user entered a positive number. Then I would jump to one of two places depending on the value of the flag:

```
jr c, Take_Care_Of_Negative_Number

jr nc, Take_Care_Of_Positive_Number
```

At least half of the ASM instructions will do things to the Z and C flags, so if you want to force values, make sure that proceeding lines of code will not mess around with the flags until you are ready to tell the calculator what to do depending on flag values.

To force the C flag to C, use the instruction SCF.

To force the C flag to NC, use the instruction OR A. This will not destroy whatever is in register A.

To force the Z flag to Z, use the instruction CP A. This will not destroy whatever is in register A.

To force the Z flag to NZ, you unfortunately have to destroy whatever is inside a register. If you don't need the value stored in register A, use OR 1. Otherwise, pick a register, set it to 0, and then use INC on it.