TIAS - a TI8x Assembler

Michael K. Pellegrino

January 14, 2021

1 Abstract

Why on Earth would I set out to create an Assembler, Disassembler, and Programs for the TI8x calculators?

As a life-long computer programmer, whenever I see something that can be programmed, I try to program it. I find it therapeutic and relaxing (just as much as programming can be frustrating and stressful). My first language was BASIC 2.0 on the Commodore-64, but soon after I learned BASIC I taught myself Assembler for the 6502. I just love(d) working with the bits and bytes, with mnemonics and opcodes, and with hexadecimal and binary. The speed at which programs would run (compared to BASIC) was mind-blowing to my ten-year-old self.

Fast-forward 40 odd years (some more odd than others) and I find myself teaching high-school math and computer programming. That's when I began using the TI8x on a daily basis. TI-Basic is ... well... okay I guess, but programs written in assembler are way cooler.

One of the only resources I could find to assemble a program that could run on the TI8x's I had was an on-line assembler. I had to write the program, assemble it online, download the .8xp file, and then load it onto my calculator. This was sort of a tedious process and if the internet was down - I would be outta luck and just have to wait it out.

I figured, why should I have to use a third-party tool to write my programs? If I were to make my own, then it would do *exactly* what I wanted - how and whenever I wanted it to. And to boot, few (if any) students would be able to modify my programs.

So I started writing a 'compiler' that would take simple commands and generate the Z80 OPCODES

for them. Of course, this wasn't good enough because the program files it generated weren't formatted with the appropriate header and checksum to run natively on the calculators.

This is where I fell into the rabbit hole. I wrote a program that created a header and checksum for my little programs and Lo! and Behold! they were able to execute on the TI8x!

My 'compiler' didn't have anything fancy, but it was a good starting point. I then started creating an assembler which grew to be the current version of TIAS.

Along the way, I felt that there were many things that I was trying to do in a program that should be part of a standard library... I'm not talking about the Native System calls... I'm talking about things like: getting user input, converting numbers into strings, and such. These things I was re-inventing every time I wrote a program it seemed. So, I started building some of that functionality into the assembler... like macros.

It's an on-going process and even though I set out to build a TI8x assembler so that I could write better programs for the TI8x's, I think that I now spend most of my time writing bits and pieces of the assembler.

So, what was the question, again? Oh - right - *Why on Earth would I set out to create an Assembler, Disassembler, and Programs for the TI8x calculators?* My answer... "Why not?".

2 Using the Assembler and Disassembler

The assembler can be executed using the following command:

```
tias input.asm output.8xp
```

Additional options: you can add a third parameter... `output.lst` that is a filename of a file that is to contain the assembler source that *was* compiled. This is for debugging purposes only.

Example: `tias input.asm output.8xp output.lst`

The **dis**assembler can be executed using the following command:

```
Example:  disassemble input.8xp
```

Additional options: use `disassemble --help` for more options

3 Built-in Functions

- loop
- pusha
- popa
- mem_clear
- user_input, (dont_)check_for_decimal_points, (dont_)check_for_negatives
- hex_input
- store_op1
- hex_to_string
- degree_mode
- radian_mode
- disp_op1

- fully_clear_screen
- convop1b (this should *probably* be `conv_op1` but let's wait until this is all finished to start messing around with the details like that.)

3.1 loop

`loop`: arguments pushed onto the stack before `call &loop`: address of code to be looped, and the number of iterations

```
Example:
    ld hl, &mycode
    push hl
    ld bc, #0x000A
    push bc
    call &loop
    ret
mycode:
    push bc
    pop bc
    ret
```

3.2 pusha

`pusha`: a macro that pushes the registers `af`, `bc`, `de`, and `hl` (in that order) onto the stack.

3.3 popa

`popa`: a macro that pops the registers `hl`, `de`, `bc`, and `af` (in that order) off of the stack.

3.4 mem_clear

`mem_clear`: zeroes out a section of memory

```
Example:

    ld hl, &start_of_region

    ld (&function_mem_clear_address), hl

    ld a, 0x05

    ld (&function_mem_clear_number_of_bytes), a

    call &mem_clear

    ret

start_of_region:

    .db 0x01

    .db 0x02

    .db 0x03

    .db 0x04

    .db 0x05

    .db 0x06; <-- this byte won't get cleared
```

3.5 user_input

`user_input`: gets a decimal value from the user and stores it in the `OP1` and `&FP_user_input`. There are 4 helper functions that go along with this function.

- `dont_check_for_decimal_points`: forces the `user_input` function to stop allowing decimal points
- `check_for_decimal_points`: forces the `user_input` function to begin allowing decimal points (this is the dafault)
- `dont_check_for_negatives`: forces the `user_input` function to stop allowing negative signs

- `check_for_negatives`: forces the `user_input` function to begin allowing negative signs (this is the dafault)

Example:

```
call &user_input
call &disp_op1
call &dont_check_for_decimal_points
call &dont_check_for_negatives
call &user_input
call &disp_op1
call &check_for_decimal_points
call &check_for_negatives
call &user_input
call &disp_op1
ret
```

3.6 hex_input

`hex_input`: gets a hexadecimal value from the user and stores it on the stack. It also puts a text string (complete with a preceding `0x`) at the address: `&function_hex_input_hex_string`.

Example:

```
call &hex_input
pop hl
bCall(DispHL)
bCall(NewLine)
ld hl, &function_hex_input_hex_string
bCall(PutS)
```

3.7 store_op1

`store_op1`: stores floating point register `OP1` as a variable on the calculator. The 1 parameter is a token (the ASCII value of the letter that is the variable) stored in `&function_store_op1_variabletoken`. Variables A-Z use the hex values 0x41 - 0x5A and θ is 0x5B.

Example:

```
call &user_input
ld a, 0x46; 0x46 is the letter F so this
          ;will store the user input in the F variable
ld (&function_store_op1_variabletoken), a
call &store_op1
ret
```

3.8 hex_to_string

`hex_to_string`: Converts a 2 byte hexadecimal value to a displayable string. The 1 parameter is a 2 byte hexadecimal word passed via the stack. The return value is the address of the string. It is returned via stack.

Example:

```
ld hl, #0xAB12
push hl
call &hex_to_string
pop hl
bCall(PutS)
;;; to print without the 0x
ld hl, #0xAB12
push hl
```

```
call &hex_to_string
pop hl
inc hl ; increase hl by 2 bytes
inc hl ; which put it past the 0x
bCall(PutS)
ret
```

3.9 degree_mode and radian_mode

`degree_mode`: puts the calculator into degree mode, and `radian_mode` puts the calculator into radian mode. In `degree_mode`, $\sin(45) = 0.7071...$ and in `radian_mode`, $\sin(\pi/4) = 0.7071....$

Example:

```
;;; find the sin(45)
call &degree_mode
ld hl, &forty_five
bCall(Mov9ToOP1)
bCall(Sin)
call &disp_op1

;;; find the sin(pi/4)
call &radian_mode
ld hl, &pi
bCall(Mov9ToOP1)
bCall(TimesPt5)
bCall(TimesPt5)
bCall(Sin)
call &disp_op1
```

```
    ret
;;; data for the example
.pi
forty_five:
    .db 0x00
    .db 0x81
    .db 0x45
    .db 0x00
    .db 0x00
    .db 0x00
    .db 0x00
    .db 0x00
    .db 0x00
```

3.10 disp_op1

`disp_op1`: Displays the value of `OP1` on the screen.

```
Example:
    ld hl, &pi
    bCall(Mov9ToOP1)
    call &disp_op1
    ret
.pi
```

3.11 fully_clear_screen

`fully_clear_screen`: Completely clears the screen. This function takes no parameters.

```
Example:

    call &fully_clear_screen

    call &disp_op1

    ret
```

3.12 convop1b

`convop1b`: Converts `OP1` into a 2 byte hexadecimal number. There is a system call provided by TI that also does this, but it has a limited input range. It is called `convop1`. This built-in function will allow all decimal values from 0 to 65535. It returns the value in `de`.

```
Example:

    call &user_input

    call &convop1b

    push de

    call &hex_to_string

    pop hl

    bCall(PutS)

    ret
```

4 Registers

- z80 16 and 8 bit registers
  - `af` (accumulator and flags): `af` cannot be used as a 16 bit register except to `push` it onto the stack, `pop` it off of the stack, and `ex` (exchange) it with the shadow register `af'`.

  - `bc`, `b`, and `c`
  - `de`, `d`, and `e`
  - `hl`, `h`, and `l`
  - `af'`: shadow register
  - `bc'`: shadow register
  - `de'`: shadow register
  - `hl'`: shadow register
  - `ix`
  - `iy`
  - `i` and `r` (not for general use)
  - `sp`: the stack pointer
- Floating Point registers
  - `OP1`
  - `OP2`
  - `OP3`
  - `OP4`
  - `OP5`
  - `OP6`

These all should be referenced by address (as in `&OP1` and `&OP4`). `Example: ld hl, &OP3` will put the *address* of `OP3` into hl.

5 Directives

As of this edition of the manual, directives must be on their own line of code

- ; Comments start with a semicolon (that is not in quotes). The Assembler does not process them.

  `Example:`

```
    ld hl, &my_string
    ; this is a comment
    bCall(PutS)
    ret
my_string:
.str ";this is NOT a comment"
```

- `.name`: This should only be in the first line of the source code, followed by a program name that is less than 9 letters long.

  If it's not given at all, the default program name is: `TI84PROG`

```
.name myprog
    ld hl, &OP2
    ld de, &some_address
    ld b, @0x09
    ldir
    ret
```

- `.dw`: Allocates memory for 1 word (2 byes) in memory and stores an immediate word there.

```
Example:
    ld hl, &my_word
    bCall(DispHL)
    ret
my_word:
.dw 0x1234
```

- `.db`: Allocates memory for 1 word (2 byes) in memory and stores an immediate byte there.

```
Example:
    ld a, (&my_byte)
    bCall(PutC)
    ret
my_byte:
```

```
.db 0x41
```

- `.str`: Allocates memory for a **null-terminated** string and stores one there.

```
Example:
    ld hl, &my_string
    bCall(PutS)
    ret
my_string:
.str "Hello World!"
```

- `.chars`: Allocates memory for a **non** null terminated string and stores one there.

```
Example:
    ld hl, &my_chars
    bCall(PutS)
    ret
my_chars:
.chars "Hello "
.chars "World!"
.db 0x00
```

- `.pi`: The floating point constant π. 9 bytes that containing the floating point value of pi. It has the label: `pi:`

```
Example:
    ld hl, &pi
    bCall(Mov9ToOP1)
    call &disp_op1
    ret
.pi
```

- `.e`: The floating point constant e. 9 bytes containing the floating point value of e. It has the label: `e:`

Example:

```
    ld hl, &e
    bCall(Mov9ToOP1)
    call &disp_op1
    ret
.e
```

- `.fp`: Allocates 9 bytes of memory and gives it a label. All bytes are `0x00`.

Example:

```
    ld hl, &FP_my_float
    bCall(Mov9ToOP1)
    call &disp_op1
    ret
.fp FP_my_float
```

6 Example Program