CN
-PRESS-

No Duh!

GUIDE TO

Z80 ASSEMBLY

by BRPXQZME

0.00	US
0.00	GB
0.00	EU
0	JP
0.00	CA

Yo, Folks. This is nothing new, but I can't put up with the best way to learn Assembly (ASMGuru, duh). The Windows Help is annoying, and too many things were un-tabbed for my taste (and, um, James is not developing it any further :( ). So all I've really done is cook up a batch of new tutorials with the same content, but better comments. If you have a problem with Adobe Acrobat, go ahead then, download AsmGuru. Unless you're some noob tryin something too hard, or you're planning on making the suckiest little programs, or going to do something totally evil to somebody's calc, nobody's trying to stop you. All my thanks go to James Matthews for writing ASMGuru, and to all his buds for helping him out.

A few (sorry, I mean a lot of) words of warning...

Understand this:

If you create a faulty program, and it screws up your calc, just remember, DON'T PANIC. Ok, you don't have the Hitchhiker's Guide, but just remember that. You don't even need to chant some sort of prayer. Geez, it's a machine for cryin out loud! You should first take all the batteries out (Yes, the backup, too). Then leave it alone overnight, then pop the batteries back in. Ok, you've cleared your RAM, and that's very disappointing the first time. But take my advice: GET OVER IT. It happens all the time. And all those games aren't really important, because that's not all there is to life. Life: If you don't have one, GET ONE! If this scared you, then ASM is not for you.

If you don't know any other programming language or you only know BASIC, ASM is not for you.

If you are planning to be evil, go sleep on a bed of nails, using a sharp rock as a pillow, until all your evil desires go away. If they don't, then ASM is not for you.

If you have an IQ below 110, then you might have a hard time with ASM, because it is VERY HARD.

You need:

A TI-83 or TI-83+ calculator. Don't worry, I'll work around the differences.

A compiler, linker, and include files. I know that was very vague, because I'm insisting that you download the Ionpak. It was pieced together by Bryan Summersett. If you like, this really has all you need to start ASM programming, but like I said, you might not be able to put up with help files. Download it off of whatever web site you got this from.

A TI-Graphlink and all the software necessary to use it.

Ion or eqivalent for the TI-83. I highly recommend MirageOS if you have an 83+

I also recommend that you get an emulator. I highly recommend Rusty Wagner's Virtual TI, since it is the only one that I know of that supports the 83+.

Oh yeah, that calc is for MATH, right? I think you should at least have a well-developed mind for math (should've passed Algebra I) because it will really help, plus I don't have any degrees for being a teacher. Hey, I'm only in middle school (but I'm GT 8-P)

Whew! That was long winded, but if you're still here, I guess you'll read the whole thing. Now about me. I'm sevvie :) as of 00-01. I've enjoyed computers since I was six. I'm in my fifth year of piano, and third of trumpet. My ICQ is 94003966, but I can assure you I'm never on :(If you are interested in contacting me, email BRPXQZME@hotmail.com and maybe I'll get to you in a few (*lets steam out of collar*) months. That's where you mail suggestions and nits. Please no viruses, flames, etc., as I don't have time for that. To mail donations to me, It's...*Uhm, don't donate anything yet, the project ain't complete yet*. Please no package bombs, Ebola, etc.

As an added bonus, when I'm done with the TI-83/+ tutorials, I'll include other systems, such as... Nintendo Game Boy! (believe it or not, the instructions are VERY similar) I will do it, but not yet (so I can keep you anxious and looking for updates at all times ;-))
NOTE: Project started Memorial Day, 2001

Tutorial I:

Beginnings

Wahoo! So lets get started!

We will be working with just 83 programming at first. Yup, no 83+. I'll show how to do that once we've learned enough. So if you have an 83+, all you can do is read along and learn, and don't try the programs on your calc. If you have an 83, just don't worry about that last bit. It was a bit bitchy when I first tried, but if you're smart you should get this.

Now, we won't do any programming yet, but you will learn the necessary ASM basic principles.

TASM

TASM is our compiler. The compiler only produces object code, which has to be linked (but you really don't have to worry about that). Ionpak has conveniently included devpac83, which will do that. Also included is asm.bat, so we won't have to worry about MS-DOS command line parameters (they can be a pain, and ASM is already hard enough).

Problem: TASM is likes periods (.) in front of its instructions, but the Ti-83 include files (.inc) that TI provided doesn't follow that convention. Thus, we either have to edit the "include" files or add some small bit of code to the beginning of every file. Editing the include files is pointless for two reasons: 1. They are frickin too long to do that, and 2. Your code won't be compatible with anyone else's include files. So, um, lets see how we work around that

About the include files

Most programming languages let you include instruction files outside of the program. Think of it as a library where common functions are kept, already written for you under some useful name that you can use in your program. For example, clearing the screen of the calc (Next tutorial) using a name instead of writing out the memory address (kinda like using a title instead of a call number). When the program is compiled and that compiler comes across a command it looks in the include file library you have told it to include ('ti83asm.inc' right now) and finds the command, then substitutes the relevant code.

Using include files is ASM

Here is your first ASM command! Muahahaha! The `#define` tag is used in the same way as C/C++ #define statement (and if you don't know what that is, GET OUTTA HERE!). Since TI's include files used `end` when TASM wants `.end` we must insert this code at the beginning of nearly all programs:

```
#define end .end
#define END .end  ; TASM is case-sensitive (somewhat)
#define equ .equ
#define EQU .equ
```

`END` and `EQU` are the most common places where this happens.

Your second instruction is ... the #include tag, same as C/C++ use. It tells the compiler what files to put in that ol "library".

Up until now, I've been ripping this off of ASMGuru, but James screwed up here. He forgot to *explain* how to use these tags. That's why it took me *three* whacks before I really understood ASM some. Well, I'll tell you now. Those four lines of code define that you're gonna use the first statement instead of the second statement. Here will be our complete header for the rest of the 83 specific tutorials.

```
.NOLIST
#define end .end
#define END .end  ; standard header
#define equ .equ
#define EQU .equ
#include "ti83asm.inc"
#include "tokens.inc"
.LIST
.org 9327h
```

As you see, the .inc files are put in the way shown above. Oh, in case you haven't noticed, anything after a semicolon is a comment (except inside a string). (NOTE: If you don't know either of those terms, please mail me your TI. You don't deserve it.) We will discuss the meaning of the code next tutorial.

Tutorial II

CALLing out and the meaning of Header

I just made that weird quote for the sake of weirdness

Today we must learn the meaning of that code. As a starting place, here's the header:

```
.NOLIST
#define end .end
#define END .end  ; standard header
#define equ .equ
#define EQU .equ
#include "ti83asm.inc"
#include "tokens.inc"
.LIST
.org 9327h
```

You ask, "Uhh...m?" Well, it's not that hard if you know.

`.NOLIST` –	Means the following will not go to the calc, it is only for TASM to see.
`#define` –	Means to use the first part instead of the second.
`#include` –	Includes the library.
`.LIST` –	Put following code in final compiled version, TASM.
`.org` –	Go to ticalc.org. Kidding. This says where in the memory the program starts. On the TI83, this should (to my knowledge) always be 9327h (h means it's in hex, although you could use $9327).
`call` --	Runs the specified romcall.

Wot's 'call'?

Wait a minute! We haven't seen *THAT* before... Well, jump in to the first full program! NOTE: I'm NOT paying for broken monitors.

```
.NOLIST
#define end .end
#define END .end
#define equ .equ
#define EQU .equ
#include “ti83asm.inc”
#include “tokens.inc”
.LIST
.org 9327h
      call _clrLCDFull  ;yes, that IS an underscore
      ret
.end
```

Obviously, `.end` ends the code, not so obviously `ret` returns you to the home screen. NOTE: You may notice that section is tabbed in. I'll talk about that later.

Compiling

Anyone ever told you, “compiling is a pain”? They’re right. That file, asm.bat, has already been made for you (luckily). Assuming you saved that file as ZCLEAR.Z80, AHEM, hint, hint. (NOTE: You could save it as ZCLEAR.ASM, also. It won’t matter, but ASM extension is also used for other assembly languages. Never have the same named Z80 and ASM file in the same folder, as it will confuse programs.) Anyway, go to the command line and type `asm ZCLEAR` and press Enter. The case should be UPPERCASE, otherwise there will be lowercase letters in your program name and your calc won’t let you use that. It will compile and link your files in one step. Nothing should be wrong at this point if you’re just cutting and pasting. Now send the program to the calc (you DO have a Graphlink, don’t you?). DO NOT try to run it. First, go to [2nd] <CATALOG> and look for the “`Send(`” thingy. Press [ENTER] (on the calc, stupid.). It should be pasted to the home screen. Press [9]. Press [PRGM] and paste prgmZCLEAR to the home screen. It should be “`Send(9prgmZCLEAR`”. Press [ENTER] and the screen will clear (unless you did something wrong). YAYY! You are technically a Starting ASM Programmer (If it worked). If it didn’t work, then YOU did something wrong (cuz every other ASM Programmer out there couldn’t possibly be wrong). Try repeating the process, see if you got a legit copy of ionpak (maybe some prankster thinks it’s funny.), and if all else fails, contact somebody else except me, because my email will be flooded with people dumber than you who didn’t listen to that last part (NUDGE, NUDGE). If you did this on the 83+, it won’t work (if it did, something is SERIOUSLY wrong).

Conclusion

Whew… You either worked your way (83) or read your way(83+) through the hardest part of Beginning ASM. Pat yourself on the back, do a victory dance, and take the 5 hour nap you deserve for this … wow, another actual ASM programmer… Until you get to the Advanced level, almost nothing is this hard.

YOU’RE STILL READING THIS!? LOOSEN UP, IT’S FIESTA TIME!!

Tutorial III

Code formatting and registers

The NEXT hardest part of ASM

Code Formatting

Lets get the easy part down first. Anything except a label or a directive must be tabbed in. You'll learn what a label is next chapter. A directive is anything with a period in front of it (most notably .org and .end). Asmguru says only that, but I also assume that anything with a # in front of it also must not be tabbed in (NOTE: If any British people get confused with the American vocabulary, a. Sorry b. No, I'm not going to make a different UK edition c. I'm glad to know that people on the other side of the Atlantic know about me, and put up with this anyway). On to the hard part: REGISTERS.

Registers

BUM BUM BUM BUMMMMMM... Even makes me look back and shiver... In fact, AW GEEZ I CAN'T EXPLAIN IT! But other people can. Here is an excerpt from Greg Parker's TI-85 ASM tutorials, which should explain it understandably.

TI-85 Assembler Programming - Data and Memory

In other programming languages, you store data in variables. You can do stuff with variables like adding two of them together and storing the result in a third variable.

In assembler, variables aren't as simple. The "variable" is actually an abstract concept. Inside the computer or calculator, variables don't exist. Instead, you directly access the memory and store your data there.

On the TI-85, "memory" means the 32K of RAM that is built-in. This memory is divided into 1-byte chunks. One byte allows you to store an eight digit binary number, so the largest number you can store in each location is a binary 11111111, or 255 in decimal. The lowest is 0. If you want to use larger numbers, you group several chunks together. One common grouping is two bytes, or a "word". Two bytes together can hold a value between 0 and 65535 (2^16).

Each one-byte chunk has a unique number, or address. We access these bytes by using the addresses, just like we use variable names to access variables. Addresses are usually given as hexadecimal numbers for convenience, so a typical address might be $80DF. Addresses used with the Z80 chip are two bytes long.

TI-85 Assembler Programming - Data and Registers

In the previous lesson, you learned about the memory, where nearly all data is stored. Unfortunately, memory access is relatively slow for the processor. So, when we want the processor to manipulate data, like add two numbers together, we don't want to do it in the memory. Instead, we load the data into special high-speed memory locations inside the processor itself called registers.

In the Z80 chip there are 14 registers. They have names instead of addresses, and are called A, F, B, C, D, E, H, L, PC, SP, IX, IY, I, and R. The one-letter ones are each one byte, and the two-letter ones are each two bytes.

For programmers, the most important register is the A register. A is also called the Accumulator. A is the focus point for nearly all data operations. For example, when we add two numbers together, one of them is stored in A, and the final answer is also stored in A.

B, C, D, E are used for temporary data, like holding the second number when adding. They are grouped in two-byte pairs - BC and DE - when we need to store larger numbers, like memory addresses.

H and L are separate registers, but they are almost always used as the two-byte pair HL and should always be used to store addresses when moving data between the memory and the registers.

The rest of the registers are unimportant for now. Some of them are used by the chip and not us. PC is the Program Counter. It stores which instruction is being executed. SP is the Stack Pointer. It is the memory address of the top of the stack. We will use the stack later, but we never actually change SP ourselves. It is the Interrupt Vector, which I won't even try to explain. The R register is the Refresh register. I won't explain it because I have no idea what it does!

F is the Flags byte. Each of its bits mean something different. For example, the Zero flag bit tells us whether the last instruction generated a zero result. The Carry bit tells us whether the last math operation required a carry. We will use those two flags later. We don't use the other flags very often.

IX is used for special purpose memory access, which we might do later. IY is used internally by the TI-85.

That's how data is stored and used on the TI-85. The addressed memory contains numbered one-byte chunks and is where data is stored. The registers are high-speed named memory locations where data can be manipulated. In the next lesson, you will learn the LD instruction, which is used to move data between registers and memory and from register to register.

TI-85 Assembler Programming - The LD instruction

Now that you know how data is stored in assembler, you can learn how to move it around. In assembler, you can't just assign any value to a memory location like you can with variables. Instead, you have to use the `LD` instruction. `LD` stands for Load. It loads data from one place into another.

There are several different ways to use the `LD` operation. The simplest one is loading a specific value into a register. Here is an example:

```
LD   A, 17
```

This gives register A the value 17. We can do the same thing with any register and any value. The value can be hexadecimal if we want.

```
LD   B, 32
LD   C, $18
LD   H, 213
```

We can do the same with any register pair:

```
LD   BC, 14238
LD   HL, $80DF
```

Another way to use `LD` is to load data from one register to another:

```
LD   B, A
```

This puts whatever value is in register `A` into register `B`. As you can see, with any use of `LD`, data is moved right to left - `17` into `A` or `A` into `B` or whatever. We can load any single register into any other register.

More complicated uses of `LD` involve moving data between the memory and the registers. As you know, the memory is accessed through addresses. The `LD` instruction is used to load data between a register and a memory location specified by an address. This address can be a raw value:

```
LD   A, ($80DF)
```

This will load the value stored at memory location $80DF and put it into register `A`. The parentheses around the number indicate the number is being used as an address, not as a value.

In the previous example, loading a single register from memory using a specific address, we have to use register `A`. This means that we cannot load any other register directly from memory. Instead, we have to load from memory into `A` and then from `A` into the register:

```
LD   A, ($80DF)
LD   C, A
```

We can also load two bytes of data from memory at once:

```
LD   BC, ($80D3)
```

This loads two bytes of data from ($80D3) into register pair `BC`. Actually, it loads from two memory locations - ($80D3) into `B` and ($80D4) into `C`. When loading a pair like this, we can use any pair we want. Another method is to use the register pair `HL` to hold the address for us:

```
LD   HL, $80DF
LD   A, (HL)
```

The first `LD` loads a value into `HL`. The second one uses that value in `HL` as an address, and loads data from that address into `A`. It has exactly the same effect as the previous `LD` example. Notice that $80DF does not have parentheses around it. In the first `LD`,

$80DF is only a value, not an address. Only in the second LD is it an address, so we put parentheses around HL to show its value is an address. We can load from memory into any register, but we cannot load into a register pair or use any other register besides HL to hold the address.

Now let's move data from the registers back into memory. The simplest way is to use an immediate value for the address:

```
LD   ($80DF), A
```

This loads the value in register A into memory location $80DF. When moving data from register into memory this way, we are restricted to using the A register. In other words, we can't load register C or any other directly into memory - we have to load the register into A, and then A into the memory.

We can also store a register pair into two memory locations:

```
LD   ($80DF), BC
```

Again, this loads into two different memory locations. We can use any register pair here.

As above, we can use a register pair to hold the address we are using:

```
LD   (HL), 123
```

This loads the value 123 into the memory location whose address is stored in HL. We can only use HL for this.

```
LD   (HL), A
LD   (HL), E
LD   (BC), A
```

Here are three examples of loading a register into memory whose address is not immediate. We can use HL to store the address and load from any register or we can use any register pair and load from A. We cannot do both, like:

```
LD   (BC), D     ; this is WRONG!!!!!
```

We are stuck with either HL for the address or A for the data.

I think that about sums up the different uses for LD. It is one of the most versatile and probably the most important instruction. There are more exceptions to most of the examples I gave, but most of them deal with the I or R or SP registers, and we won't be dealing with them. If you really want to know, find a Z80 programming book and look it up yourself.

>>Back to ME. I could have rewritten this, but THANX, GREG YOU'RE A LIFESAVER! Only after reading this did I finally understand registers, so I made you do it, too. I did not rewrite this, so then I could give thanks to Greg (whoever you are).

NEXT: Displaying Text.