# eZ80 Heaven

*Release 1.0*

**KingInfinity and HactarCE**

**Apr 01, 2017**

# Tutorials

Contents:

# CHAPTER 1

# Introduction to Assembly

You want more games for your shiny new TI-84+CE? Why don't you make them yourself?

## What is assembly?

Assembly is a very powerful language that you can use on your TI-84+CE! When you write in assembly, you write in a human readable form of machine code, which allows you to create quicker programs that can access and do more than TI-BASIC. Mistakes in assembly can be messier than in TI-BASIC- but usually only as severe as a ram clear, so before you start this guide, back up your calculator to prevent any data loss!

## Setting up

### What you'll need

#### A program editor

To write your program, you can use almost any text editor. We prefer Notepad++, as it is really easy to use for beginners and it has some nice features for developing.

Download Here: Notepad++
Syntax Highlighting: Notepad++ eZ80 Syntax Highlighting

#### An assembler

An assembler lets you take the code you have created and make it into a program that the TI-84 Plus CE can use. We recommend SPASM-ng when coding in eZ80.

Download Here: Spasm-ng

### A debugger/emulator or a TI-84+CE

A debugger is useful for seeing what your code does and finding errors. Unfortunately, at the time of this writing, the only 84+CE emulator available for download is CEmu, which is hardly in a usable state. Because of this, we need TI's Connect software to transfer our programs to our calculator, and we'll also need a link cable (which should have come with your calculator) and the calculator itself.

Download Here: TI Connect CE

### TI 84+CE Equates File

This file tells your program and the compiler where things are located on your calculator.

Download Here: ti84pce.inc

## Setting up the assembler

Alright, now we can get to putting together all the materials we just downloaded! Create a folder and call it "My First ASM Program" (or anything else). Inside create three folders: `bin`, `includes`, and `tools`. Rename `SPASM` to just `spasm.exe` and put it inside the folder named tools. Put ti84pce.inc inside the includes folder.

Create a file named `Example.asm` in your folder. Inside this file, write the following code.

```
#include "includes\ti84pce.inc"

 .assume ADL=1
 .org userMem-2
 .db tExtTok,tAsm84CeCmp

 call _homeup
 call _ClrScrnFull
 ld hl,TutorialText
 call _PutS
 call _GetKey
 call _ClrScrnFull
 res donePrgm,(iy+doneFlags)
 ret

TutorialText:
  .db "Excellent job! :) You havecreated your first assembly program!",0
```

*Don't worry; we'll revisit what all this code does later on.*

Now, create a file named `build.bat`. Inside this file, write the following code:

```
@echo off
:A
tools\spasm -E -T Example.asm bin\Example.8xp
Pause
Goto A
```

Run `build.bat` by double-clicking on it. Inside the "bin" folder you should see a file named FIRSTPRGM.8xp. Transfer this onto your calculator using TI Connect CE and run it by pressing `2nd` → `Catalog` and selecting the `Asm(` token, then pressing `Prgm` and selecting `EXAMPLE` (or whatever you program was called). It should look something like this: `Asm(prgmEXAMPLE`

If this doesn't work, try again to make sure you followed each step of the tutorial again. You can also post for help on the Cemetech forums.

*In the next lesson we'll dive right into learning assembly.*

CHAPTER 2

# Program Structure

Every program in assembly follows a specific format.

```
#include "includes\ti84pce.inc"
.assume ADL=1
.org userMem-2
 .db tExtTok,tAsm84CeCmp

 ; Program code

 ret
```

But what does all of this code do?

```
#include "includes\ti84pce.inc"
```

This is a file that includes lots of locations of routines and variables and various other things in the calculator's operating system that we can use.

```
.assume ADL=1
```

This tells the assembler to assume that ADL mode is on. ADL mode means the processor is using 24-bit register pairs and memory addresses, which is the the calculator's default.

```
.org userMem-2
```

This tells the assembler that this program will be located at `userMem-2`, so any memory addresses we reference should be relative to that. (The program is actually placed at `userMem`, but there are two bytes right after this which are removed from the program by the OS before it is placed at `userMem`.)

```
.db tExtTok,tAsm84CeCmp
```

This is exactly equivalent to the `Asm84CEPrgm` token in BASIC; it tells your calculator that it should be executing an assembly program instead of a BASIC program. This is why when you try to run an assembly program on your calculator without the `Asm(` token, you get an error (and why you can't run a BASIC program with the `Asm(` token).

```
; Program code
```

This is where you can actually write the program's code.

```
ret
```

This exits the program (usually). More on that later.

*In the next tutorial we'll be learn about how numbers work on your calculator.*

CHAPTER 3

# Hex, Binary, and Decimal

**If you are already familiar with binary and hexadecimal, you may skip straight on to the next tutorial.**

If you would prefer a video tutorial, you can use this instead. Just know that assembly programmers use the prefix "$" for hexadecimal and "%" for binary.

When we humans write numbers, we like to use base 10, also called decimal. Decimal is a number system in which there are ten digits (0-9). We can write any number from zero to nine using that digit.

- 4

But if we want to write a number greater than nine, we have to use two digits.

- 42

The digit on the left tells you how many "tens" there are in this number (because we use base 10) and the digit on the right tells you how many "ones" there are in this number. So we can say that a two-digit number is equal to `( 10 * digit on the left ) + digit on the right`.

For numbers greater than 99, the highest two-digit number, we use three digits.

- 256

The digit on the right now tells you how many "hundreds" there are (one hundred being the next power of ten), and the middle digit tells you how many "tens" there are, etc. So we can say that a three-digit number is equal to "( 100 * digit on the left ) + ( 10 * digit in the middle ) + digit on the right

I'm sure you can see the pattern by now.

Humans may like base 10, but computers don't. Computers can't keep track of ten digits very easily. They only like two: 0 and 1. A number system using only two digits is base 2, or binary. When we write a binary number, we usually put a percent sign `%` in front of it to make it clear that it isn't base 10. Here's what a typical binary number may look like: %101010

In base 10, each digit is ten times as significant as the next. For example, a small change in the "hundreds" place makes a much bigger change in the value of a number than a change in the "tens" place, and likewise between the "tens" and "ones" places. In binary, each digit is twice as significant as the one after it. What do I mean by that? Well on the far left, there's the "ones" place as we would expect it. After that comes the "twos" place. And after that comes the “fours” place (2 * 2 = 4). And the “eights” (4 * 2 = 8), “sixteens” (8 * 2 = 16), “thirty-twos,” and so on. It will really pay to memorize your powers of two at least up to 2^8, which is 256 (in decimal notation).

So how can we use this information to go from a binary number to a decimal number? We make a chart!

| Place | 32’s | 16’s | 8’s | 4’s | 2’s | 1’s |
|---|---|---|---|---|---|---|
| Digit | 1 | 0 | 1 | 0 | 1 | 0 |

Okay, so now how do we do this? We multiply down and add across. So let’s start from the right:

There is a 0 in the 1s place, so we multiply 1 times 0. The result is 0.

There is a 1 in the 2s place, so we multiply 2 times 1. The result is 2.

There is a 0 in the 4s place, so we multiply 4 times 0. The result is 0.

There is a 1 in the 8s place, so we multiply 8 times 1. The result is 8.

There is a 0 in the 16s place, so we multiply 16 times 0. The result is 0.

There is a 1 in the 32s place, so we multiply 32 times 1. The result is 32.

Now we add the results. So 32 + 0 + 8 + 0 + 2 + 0 = 42. %101010 = 42. This may seem like a rather tedious way to work with numbers, stretching what would otherwise be a relatively short and easy-to-read decimal number into a long, inefficient, and annoying string of binary, which is why we programmers have another way of writing numbers: hexadecimal!

Hexadecimal, often abbreviated to hex, is precisely what the name suggests: base 16. Hexadecimal numbers are much shorter than binary, and usually shorter than decimal too. In assembly, we put a dollar sign `$` in front of a number to make it clear that the number is hexadecimal. (Sometimes you will see `0x` in front of a hexadecimal number too.) Hexadecimal has a “ones” place, a “sixteens” place, (bear with me here) a “256s” place (16 * 16 = 256), a “65536s” place (256 * 16 = 65536), a “16777216s” place (65536 * 16 = 16777216), and so on. It gets really big really quickly.

But wait! In decimal, we had ten digits. Binary had two. Hexadecimal needs... sixteen digits, right? How do we even write that? Well, we use letters. In Hexadecimal, when a digit is ten, we use the letter A. When a digit is eleven, we use the letter B. C is twelve, D is thirteen, E is fourteen, and F is fifteen. (Anything past that spills over into the next digit, just like how decimal doesn’t have anything over the digit 9.)

Let’s use another chart to convert from hexadecimal to decimal.

| Place | 16384’s | 256’s | 16’s | 1’s |
|---|---|---|---|---|
| Digit | 9 | 9 | 2 | C |

Just like with binary numbers, we multiply down and add across. We just have to remember that letters A-F correspond to the numbers 10-15.

There is a C in the 1s place, so we multiple 1 times 12 (C = 12). The result is 12.

There is a 2 in the 16s place, so we multiple 2 times 16. The result is 32.

There is a 9 in the 256s place, so we multiple 2 times 16. The result is 2304.

There is a 9 in the 65536s place, so we multiple 2 times 16. The result is 36864.

36864 + 2304 + 32 + 12 = 39212. That was a very large number. If we were to write it out in binary it would be %1001100100101100. Ick. Hexadecimal makes things a lot cleaner. And the best part is that every four digits in binary is equal to EXACTLY one hexadecimal digit! So if you split up that icky binary string into sections of four (%1001 %1001 %0010 %1100) every little section corresponds with one of the digits of the hexadecimal string. This makes converting between binary and hexadecimal much easier if you just memorize which 4-digit binary numbers go with which hexadecimal digits.

Okay, so everything in computers is actually binary, which can be represented more nicely in hexadecimal. But even a computer can't manage a humongous amount of binary in one large lump. It needs to organize things. So it splits up the data into sections called bytes. Every binary digit in a computer is a bit. If the bit is one, we say it is on. If it is zero, we say it is off. Every byte has 8 bits. That means each byte is an 8-digit binary number, or a 2-digit hexadecimal number. Since a single byte is two hexadecimal digits and the highest number you can write with two hexadecimal digits is $FF (255 in decimal), a byte can store a number from 0 to 255. This is why lots of computer things have limits at 255 or 256. You know the term kilobyte, which is 1024 bytes (roughly a thousand). A megabyte is of course 1048576 bytes (roughly a million).

Finally, computers start counting with zeroes, so if I have two files, one would be labeled file 0, and the other file 1.

Here are some practice problems to do:

| Decimal | Hexadecimal | Binary |
|---|---|---|
| 10 | ? | ? |
| 72 | $48 | ? |
| ? | ? | 101011 |
| ? | $1E | ? |

If you can do all of these problems, you are ready to move on to the next tutorial.

If you need more help, visit the additional learning page: here

We'll be using these concepts very frequently, so make sure you have mastered them before moving on to the next tutorial.

*In the next tutorial we'll be learning about how information is stored on your calculator.*

CHAPTER 4

# Registers and Indirection

## The Registers

If you've ever programmed in a different language, you've used something called variables to store your data temporarily. In assembly, we use registers.

There are many registers you can use in assembly. The ones we will be covering today are: A, B, C, D, E, F, H, and L, which can be paired together to form AF, BC, DE, and HL. As you probably have guessed, when a register is paired with another it can hold more information.

Recall from the last tutorial bits and bytes? H is a 1 byte register, or 8 bits, while HL is a 3 byte register pair, or 24 bits.

Now that we know about registers, it's time we learn our first instruction!

```
ld destination,source
```

What does this code do? It puts the value of source into the given destination. We can use registers as arguments for this instruction:

```
ld h,d
```

Puts the content of D inside H.

We can also use numbers:

```
ld hl,7
```

Puts 7 into HL.

Remember the size of the registers you are using:

```
ld hl,a
```

This not a valid instruction, as both arguments have to be 8 bit registers.

## Addresses

To fully understand what indirection is, we have to know what an address is. An address is a number that corresponds to a specific byte in RAM. Add one to the address and you go forward a byte; subtract one and you go backward a byte.

You can use addresses in place of registers with the LD instruction

```
ld hl,(133212)
```

This will put whatever is inside address 133212, the 133213th byte, and the 2 bytes after that (hl is 3 bytes), inside the hl register pair.

But why do we have parenthesis around the address? This is something called indirection.

## Indirection

*Note: Indirection is slightly hard to grasp for newcomers, and we'll touch more on this later, so just do your best*

Indirection tells you whether or not you are using the number as an address, or just as a number. Here the example from above:

```
ld hl,(133212)
```

How does the program know whether you are talking about the 133213th byte or the actual number 133212? The parenthesis tell the program to use the number as an address.

```
ld hl,133212 ;let's put 133212 inside hl. Note, there are no parenthesis, so we're␣
↪talking about the number 133212, and not the 133213th byte.
ld a,2 ;also no parenthesis, we mean 2
ld (hl),a ;we can decide later to use it as an address, by putting the parenthesis␣
↪around it. Now we are putting the value of A, 2, inside the address stored in hl.
```

Don't worry if you don't immediately get it, it will come to you eventually. Now it's your turn to try

```
ld de,(133215)
ld (132918),de
```

If you know what's happening here, good for you! If you don't, it's ok. You'll get it later.

*In the next tutorial we will learn about labels.*

CHAPTER 5

# Labels

Labels consist of an identifier (or symbol) followed by a colon `:`. The identifier can contain numbers, letters, and underscores _, but cannot start with a number. Labels must be defined only once.

When a program is compiled, each label is replaced with a 24bit address. A label can be used to identify a string of data, a location to jump to, or a subroutine to call.

From the example program:

```
.nolist
#include "includes\ti84pce.inc"
.list
.assume ADL=1
.org userMem-2
.db tExtTok,tAsm84CeCmp
 call _homeup
 call _ClrScrnFull
 ld hl,TutorialText
 call _PutS
 call _GetKey
 call _ClrScrnFull
 res donePrgm,(iy+doneFlags)
 ret
TutorialText:
 .db "Excellent job! :) You havecreated your first assembly program!",0
```

The label `TutorialText` is used by the compiler to store the location of the start of the string. (`TutorialText` now refers to the 24bit address of the start of the string.)

Here, the value of the label `TutorialText` is loaded into hl (the address of the start of the string) and `call _PutS` displays the string located at the address in hl. `_PutS` itself is also a label (defined in `ti84pce.inc`) that refers to the address of a subroutine built into TI-OS.

# CHAPTER 6

# Romcalls

Romcalls are pieces of code that run system predefined functions.

A list of romcalls can be found here, with one problem. Since this list is for the TI-83 Plus, some of the romcalls are missing or outdated. Also, the romcalls on the +CE are prefixed with a single underscore.

How does the assembler know where the romcalls on the TI-84+CE are? Surely they must be in a different location then the CE? The answer lies in the .inc file that you include at the start of every program.

```
#include "includes\ti84pce.inc"
```

Inside is a list of addresses that define where the romcalls (and other important items) are located. Each calculator has a different .inc file.

So how do we call a romcall?

```
call _PutS
```

This will call the romcall _PutS, which as you can see in the system routines pdf, reads the string inside the address in HL.

Recall from the very first tutorial the example program

```
.nolist
#include "includes\ti84pce.inc"
.list
.assume ADL=1
.org userMem-2
.db tExtTok,tAsm84CeCmp
 call _homeup
 call _ClrScrnFull
 ld hl,TutorialText
 call _PutS
 call _GetKey
 call _ClrScrnFull
 res donePrgm,(iy+doneFlags)
 ret
```

```
TutorialText:
 .db "Excellent job! :) You havecreated your first assembly program!",0
```

Now that we know romcalls, we can figure out what each line does.

```
call _homeup
call _ClrScrnFull
```

_HomeUp isn't in the System Routines file, but it and _ClrScrnFull set up for the text output by resetting the screen.

```
ld hl,TutorialText
call _PutS
```

From the last tutorial, we learned about labels, and how they are converted to addresses at runtime. Since _PutS reads from HL as an address, we can just put the address number, and not the contents inside HL. This means we do not use indirection and we do not need parenthesis.

_PutS reads a null terminating string from HL. All that means is that the string ends in 0.

```
call _GetKey
call _ClrScrnFull
res donePrgm,(iy+doneFlags)
ret
```

We call _GetKey to pause the program until a key is pressed. Afterword, we clear the screen again using _ClrScrnFull. Don't worry about the second to last line for now, we'll learn what it does later. Finally, the program ends.

Once again, you can learn more about each romcall in the PDF above. From now on, if there is a romcall you don't recognize, use the PDF.

*The next tutorial's contents are unknown. How mysterious!*

CHAPTER 7

# Hooks

CHAPTER 8

# Arithmetic Instructions

## ADC

**ADC M,N** Add with Carry

**Description**

Adds `M`, `N`, and the carry flag (+0 or +1) and stores the result in `M`.

`M += N + Carry`

**Uses**

- Storing the state of the carry flag in either `A` or `HL` with `adc a,0` or `ld bc/de,0 \ adc hl,bc/de`
- 16 bit addition which checks if the result is zero (see Notes)
- Can be useful otherwise in very specific situations

**Results**

| Register/Flag | 8-bit | 16-bit (non-ADL) | 24-bit (ADL) |
|---|---|---|---|
| `M` | Set to the result of `M + N + Carry` | | |
| `S` flag | Set if the result is negative; else reset | | |
| `Z` flag | Set if the result is 0; else reset | | |
| `H` flag | Set if carry from bit 3; else reset | Set if carry from bit 11; else reset | |
| `P/V` flag | Set if overflow; else reset | | |
| `N` flag | Reset | | |
| `C` flag | Set if carry from bit 7; else reset | Set if carry from bit 15; else reset | Set if carry from bit 23; else reset |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| adc a,a | $8F | 1F | X | X |
| adc a,b | $88 | 1F | X | X |
| adc a,c | $89 | 1F | X | X |
| adc a,d | $8A | 1F | X | X |
| adc a,e | $8B | 1F | X | X |
| adc a,h | $8C | 1F | X | X |
| adc a,l | $8D | 1F | X | X |
| adc a,ixh | $DD, $8C | 2F | X | X |
| adc a,ixl | $DD, $8D | 2F | X | X |
| adc a,iyh | $FD, $8C | 2F | X | X |
| adc a,iyl | $FD, $8D | 2F | X | X |
| adc a,(hl) | $8E | 1F + 1R | 2F + 1R | 2F + 1R |
| adc a,(ix+n) | $DD, $8E, n | 3F + 1R | 4F + 1R | 4F + 1R |
| adc a,(iy+n) | $FD, $8E, n | 3F + 1R | 4F + 1R | 4F + 1R |
| adc a,n | $CE, n | 2F | X | X |
| adc hl,bc | $ED, $4A | 2F | 3F | 3F |
| adc hl,de | $ED, $5A | 2F | 3F | 3F |
| adc hl,hl | $ED, $6A | 2F | 3F | 3F |
| adc hl,sp | $ED, $7A | 2F | 3F | 3F |

**Notes**

- Unlike ADD, this instruction **does not** support using the index registers IX and IY as the first operand.
- Unlike ADD, this instruction **does** modify the `Z` flag when doing 16-bit and 24-bit addition.

**See Also** ADD, SBC, SUB

## ADD

**ADD M,N** Add

**Description**

Adds `M` and `N` and stores the result in `M`.

`M += N`

**Results**

| Register/Flag | 8-bit | 16-bit (non-ADL) | 24-bit (ADL) |
|---|---|---|---|
| `M` | Set to the result of `M + N` | | |
| `S` flag | Set if the result is negative; else reset | Not affected | |
| `Z` flag | Set if the result is 0; else reset | Not affected | |
| `H` flag | Set if carry from bit 3; else reset | Set if carry from bit 11; else reset | |
| `P/V` flag | Set if overflow; else reset | Not affected | |
| `N` flag | Reset | | |
| `C` flag | Set if carry from bit 7; else reset | Set if carry from bit 15; else reset | Set if carry from bit 23; else reset |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| add a,a | $87 | 1F | X | X |
| add a,b | $80 | 1F | X | X |
| add a,c | $81 | 1F | X | X |
| add a,d | $82 | 1F | X | X |
| add a,e | $83 | 1F | X | X |
| add a,h | $84 | 1F | X | X |
| add a,l | $85 | 1F | X | X |
| add a,ixh | $DD, $84 | 2F | X | X |
| add a,ixl | $DD, $85 | 2F | X | X |
| add a,iyh | $FD, $84 | 2F | X | X |
| add a,iyl | $FD, $85 | 2F | X | X |
| add a,(hl) | $86 | 1F + 1R | 2F + 1R | 2F + 1R |
| add a,(ix+n) | $DD, $86, n | 3F + 1R | 4F + 1R | 4F + 1R |
| add a,(iy+n) | $FD, $86, n | 3F + 1R | 4F + 1R | 4F + 1R |
| add a,n | $C6, n | 2F | X | X |
| add hl,bc | $0A | 1F | 2F | 2F |
| add hl,de | $1A | 1F | 2F | 2F |
| add hl,hl | $2A | 1F | 2F | 2F |
| add hl,sp | $3A | 1F | 2F | 2F |
| add ix,bc | $DD, $0A | 2F | 3F | 2F |
| add ix,de | $DD, $1A | 2F | 3F | 3F |
| add ix,ix | $DD, $2A | 2F | 3F | 3F |
| add ix,sp | $DD, $3A | 2F | 3F | 3F |
| add iy,bc | $FD, $0A | 2F | 3F | 3F |
| add iy,de | $FD, $1A | 2F | 3F | 3F |
| add iy,iy | $FD, $2A | 2F | 3F | 3F |
| add iy,sp | $FD, $3A | 2F | 3F | 3F |

**Notes**

- Unlike ADC, this instruction **does** support using the index registers IX and IY as the first operand. This is the only difference in the allowed instructions between the two.
- This instruction **does not** modify the `S`, `Z`, and `P/V` flags when doing 16-bit and 24-bit addition. For that, you can use ADC.
- Like most instructions, HL, IX, and IY are mutually exclusive. I.e. they cannot be used in the same instructions.

**See Also** ADC, SBC, SUB

## CP

**CP M,N** Compare

**Description**

Subtracts `N` from `M` (and updates the flags accordingly), but doesn't modify either operand.

`M - N`

**Uses**

- Comparing two bytes

**Results**

| Register/Flag | 8-bit |
|---|---|
| `S` flag | Set if the result is negative; else reset |
| `Z` flag | Set if `M` = `N` |
| `H` flag | Set if borrow from bit 4; else reset |
| `P/V` flag | Set if overflow; else reset |
| `N` flag | Set |
| `C` flag | Set if borrow; else reset |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| cp a,a | $BF | 1F | X | X |
| cp a,b | $B8 | 1F | X | X |
| cp a,c | $B9 | 1F | X | X |
| cp a,d | $BA | 1F | X | X |
| cp a,e | $BB | 1F | X | X |
| cp a,h | $BC | 1F | X | X |
| cp a,l | $BD | 1F | X | X |
| cp a,ixh | $DD, $BC | 2F | X | X |
| cp a,ixl | $DD, $BD | 2F | X | X |
| cp a,iyh | $FD, $BC | 2F | X | X |
| cp a,iyl | $FD, $BD | 2F | X | X |
| cp a,(hl) | $BE | 1F + 1R | 2F + 1R | 2F + 1R |
| cp a,(ix+n) | $DD, $BE, n | 3F + 1R | 4F + 1R | 4F + 1R |
| cp a,(iy+n) | $FD, $BE, n | 3F + 1R | 4F + 1R | 4F + 1R |
| cp a,n | $FE, n | 2F | X | X |

**Notes**

- **When using CP with signed integers, the flags are set as follows:**
  - `Z` means `M` = `N`
  - `NZ` means `M` `N`
  - `C` means `M` < `N`
  - `NC` means `M` `N`

**See Also** SUB, SBC, CPI, CPIR, CPD, CPDR

## DAA

**DAA** Decimal Adjust Accumulator

**Description**

Does cool stuff with binary-coded decimal.

**Uses**

- Probably useful for BCD arithmetic?

**Results**

| Register/Flag | 8-bit |
|---|---|
| `S` flag | Set to the 7th bit of the result |
| `Z` flag | Set if the result is 0; else reset |
| `H` flag | Really complicated |
| `P/V` flag | Set if the result has even parity; else reset |
| `N` flag | Not affected |
| `C` flag | Really complicated |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| daa | $27 | 1F | X | X |

**Notes**

- **When using CP with signed integers, the flags are set as follows:**
  - `Z` means `M` = `N`
  - `NZ` means `M` `N`
  - `C` means `M` < `N`
  - `NC` means `M` `N`

**See Also** ADD, ADC, RLD, RRD, SBC, SUB

## DEC

**DEC M** Decrement

**Description**

Decrements `M` by 1.

`M -= 1`

**Results**

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| | | | | |

dec a $3D 1F X dec b $05 1F X dec c $0D 1F X dec d $15 1F X dec e $1D 1F X dec h $25 1F X dec l $2D 1F X dec ixh $DD, $25 2F X dec ixl $DD, $2D 2F X dec iyh $FD, $25 2F X dec iyl $FD, $2D 2F X dec (hl) $86 1F + 1R + 1W + 1 2F + 1R + 1W + 1 dec (ix+n) $DD, $35, n 3F + 1R + 1W + 1 4F + 1R + 1W + 1 dec (iy+n) $FD, $35, n 3F + 1R + 1W + 1 4F + 1R + 1W + 1 dec bc $ED, $0B 1F 2F dec de $ED, $1B 1F 2F dec hl $ED, $2B 1F 2F dec sp $ED, $3B 1F 2F dec ix $DD, $2B 2F 3F dec iy $ED, $2B 2F 3F ================ ================ ================ ================ ================

**Notes**

- While this instruction can be combined with a `jr nz,LoopLabel` to create a great 8-bit loop, using `dec` with a 16-bit or 24-bit operand will not alter the flags, so you'll need another method of checking if the register pair is zero.

**See Also** INC

## INC

**INC M** Increment

**Description**

Increments `M` by 1.

`M += 1`

**Results**

| Register/Flag | 8-bit | 16-bit (non-ADL) | 24-bit (ADL) |
|---|---|---|---|
| `M` | Set to the result of `M + 1` | | |
| `S` flag | Set if the result is negative; else reset | Not affected | |
| `Z` flag | Set if the result is 0; else reset | Not affected | |
| `H` flag | Set if carry from bit 3; else reset | Not affected | |
| `P/V` flag | Set if `M` was $7F before operation; else reset | Not affected | |
| `N` flag | Reset | Not affected | |
| `C` flag | Not affected | | |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| inc a | $3C | 1F | X | X |
| inc b | $04 | 1F | X | X |
| inc c | $0C | 1F | X | X |
| inc d | $14 | 1F | X | X |
| inc e | $1C | 1F | X | X |
| inc h | $24 | 1F | X | X |
| inc l | $2C | 1F | X | X |
| inc ixh | $DD, $24 | 2F | X | X |
| inc ixl | $DD, $2C | 2F | X | X |
| inc iyh | $FD, $24 | 2F | X | X |
| inc iyl | $FD, $2C | 2F | X | X |
| inc (hl) | $34 | 1F + 1R + 1W + 1 | 2F + 1R + 1W + 1 | 2F + 1R + 1W + 1 |
| inc (ix+n) | $DD, $34, n | 3F + 1R + 1W + 1 | 4F + 1R + 1W + 1 | 4F + 1R + 1W + 1 |
| inc (iy+n) | $FD, $34, n | 3F + 1R + 1W + 1 | 4F + 1R + 1W + 1 | 4F + 1R + 1W + 1 |
| inc bc | $ED, $0B | 1F | 2F | 2F |
| inc de | $ED, $1B | 1F | 2F | 2F |
| inc hl | $ED, $23 | 1F | 2F | 2F |
| inc sp | $ED, $3B | 1F | 2F | 2F |
| inc ix | $DD, $23 | 2F | 3F | 3F |
| inc iy | $ED, $23 | 2F | 3F | 3F |

**Notes**

- Keep in mind that the flags are not altered when using this instruction with a 16-bit and 24-bit operand.

**See Also** DEC

CHAPTER 9

# Bit Manipulation

## BIT

**BIT b,M** Test Bit

**Description**

Checks the `b``th bit of ``M` and stores the inverse in the `Z` flag. `b` must be a hard-coded number between 0 and 7.

**Results**

| Register/Flag | 8-bit | 16-bit (non-ADL) | 24-bit (ADL) |
|---|---|---|---|
| `S` flag | Undefined | | |
| `Z` flag | Set if bit `b` of `M` is zero | | |
| `H` flag | Set | | |
| `P/V` flag | Undefined | | |
| `N` flag | Reset | | |
| `C` flag | Not affected | | |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| bit 0,a | $CB, $47 | 2F | X | X |
| bit 0,b | $CB, $40 | 2F | X | X |
| bit 0,c | $CB, $41 | 2F | X | X |
| bit 0,d | $CB, $42 | 2F | X | X |
| bit 0,e | $CB, $43 | 2F | X | X |
| bit 0,h | $CB, $44 | 2F | X | X |
| bit 0,l | $CB, $45 | 2F | X | X |
| bit 0,(hl) | $CB, $46 | 2F + 1R | 3F + 1R | 3F + 1R |
| bit 0,(ix+n) | $DD, $CB, n, $46 | 4F + 1R | 5F + 1R | 5F + 1R |
| bit 0,(iy+n) | $FD, $CB, n, $46 | 4F + 1R | 5F + 1R | 5F + 1R |
| bit 1,a | $CB, $4F | 2F | X | X |
| bit 1,b | $CB, $48 | 2F | X | X |

Continued on next page

Table 9.1 – continued from previous page

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| bit 1,c | $CB, $49 | 2F | X | X |
| bit 1,d | $CB, $4A | 2F | X | X |
| bit 1,e | $CB, $4B | 2F | X | X |
| bit 1,h | $CB, $4C | 2F | X | X |
| bit 1,l | $CB, $4D | 2F | X | X |
| bit 1,(hl) | $CB, $4E | 2F + 1R | 3F + 1R | 3F + 1R |
| bit 1,(ix+n) | $DD, $CB, n, $4E | 4F + 1R | 5F + 1R | 5F + 1R |
| bit 1,(iy+n) | $FD, $CB, n, $4E | 4F + 1R | 5F + 1R | 5F + 1R |
| bit 2,a | $CB, $57 | 2F | X | X |
| bit 2,b | $CB, $50 | 2F | X | X |
| bit 2,c | $CB, $51 | 2F | X | X |
| bit 2,d | $CB, $52 | 2F | X | X |
| bit 2,e | $CB, $53 | 2F | X | X |
| bit 2,h | $CB, $54 | 2F | X | X |
| bit 2,l | $CB, $55 | 2F | X | X |
| bit 2,(hl) | $CB, $56 | 2F + 1R | 3F + 1R | 3F + 1R |
| bit 2,(ix+n) | $DD, $CB, n, $56 | 4F + 1R | 5F + 1R | 5F + 1R |
| bit 2,(iy+n) | $FD, $CB, n, $56 | 4F + 1R | 5F + 1R | 5F + 1R |
| bit 3,a | $CB, $5F | 2F | X | X |
| bit 3,b | $CB, $58 | 2F | X | X |
| bit 3,c | $CB, $59 | 2F | X | X |
| bit 3,d | $CB, $5A | 2F | X | X |
| bit 3,e | $CB, $5B | 2F | X | X |
| bit 3,h | $CB, $5C | 2F | X | X |
| bit 3,l | $CB, $5D | 2F | X | X |
| bit 3,(hl) | $CB, $5E | 2F + 1R | 3F + 1R | 3F + 1R |
| bit 3,(ix+n) | $DD, $CB, n, $5E | 4F + 1R | 5F + 1R | 5F + 1R |
| bit 3,(iy+n) | $FD, $CB, n, $5E | 4F + 1R | 5F + 1R | 5F + 1R |
| bit 4,a | $CB, $67 | 2F | X | X |
| bit 4,b | $CB, $60 | 2F | X | X |
| bit 4,c | $CB, $61 | 2F | X | X |
| bit 4,d | $CB, $62 | 2F | X | X |
| bit 4,e | $CB, $63 | 2F | X | X |
| bit 4,h | $CB, $64 | 2F | X | X |
| bit 4,l | $CB, $65 | 2F | X | X |
| bit 4,(hl) | $CB, $66 | 2F + 1R | 3F + 1R | 3F + 1R |
| bit 4,(ix+n) | $DD, $CB, n, $66 | 4F + 1R | 5F + 1R | 5F + 1R |
| bit 4,(iy+n) | $FD, $CB, n, $66 | 4F + 1R | 5F + 1R | 5F + 1R |
| bit 5,a | $CB, $6F | 2F | X | X |
| bit 5,b | $CB, $68 | 2F | X | X |
| bit 5,c | $CB, $69 | 2F | X | X |
| bit 5,d | $CB, $6A | 2F | X | X |
| bit 5,e | $CB, $6B | 2F | X | X |
| bit 5,h | $CB, $6C | 2F | X | X |
| bit 5,l | $CB, $6D | 2F | X | X |
| bit 5,(hl) | $CB, $6E | 2F + 1R | 3F + 1R | 3F + 1R |
| bit 5,(ix+n) | $DD, $CB, n, $6E | 4F + 1R | 5F + 1R | 5F + 1R |
| bit 5,(iy+n) | $FD, $CB, n, $6E | 4F + 1R | 5F + 1R | 5F + 1R |
| bit 6,a | $CB, $77 | 2F | X | X |

Continued on next page

Table 9.1 – continued from previous page

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| bit 6,b | $CB, $70 | 2F | X | X |
| bit 6,c | $CB, $71 | 2F | X | X |
| bit 6,d | $CB, $72 | 2F | X | X |
| bit 6,e | $CB, $73 | 2F | X | X |
| bit 6,h | $CB, $74 | 2F | X | X |
| bit 6,l | $CB, $75 | 2F | X | X |
| bit 6,(hl) | $CB, $76 | 2F + 1R | 3F + 1R | 3F + 1R |
| bit 6,(ix+n) | $DD, $CB, n, $76 | 4F + 1R | 5F + 1R | 5F + 1R |
| bit 6,(iy+n) | $FD, $CB, n, $76 | 4F + 1R | 5F + 1R | 5F + 1R |
| bit 7,a | $CB, $7F | 2F | X | X |
| bit 7,b | $CB, $78 | 2F | X | X |
| bit 7,c | $CB, $79 | 2F | X | X |
| bit 7,d | $CB, $7A | 2F | X | X |
| bit 7,e | $CB, $7B | 2F | X | X |
| bit 7,h | $CB, $7C | 2F | X | X |
| bit 7,l | $CB, $7D | 2F | X | X |
| bit 7,(hl) | $CB, $7E | 2F + 1R | 3F + 1R | 3F + 1R |
| bit 7,(ix+n) | $DD, $CB, n, $7E | 4F + 1R | 5F + 1R | 5F + 1R |
| bit 7,(iy+n) | $FD, $CB, n, $7E | 4F + 1R | 5F + 1R | 5F + 1R |

**Notes**

- Interestingly enough, the index registers `IXH`, `IXL`, `IYH`, and `IYL` cannot be used in this instruction, yet `(IX+n)` and `(IY+n)` are allowed as operands.
- **The `Z` flag is set as follows:**
  - `Z` means the bit is **zero**
  - `NZ` means the bit is **one** (non-zero)

**See Also** RES, SET

## RES

**RES b,M** Reset Bit

**Description**

Sets the `b``th bit of ``M` to zero. `b` must be a hard-coded number between 0 and 7.

**Results**

| Register/Flag | 8-bit | 16-bit (non-ADL) | 24-bit (ADL) |
|---|---|---|---|
| `M` | The ‘‘b‘‘th bit of M is set to zero | | |
| `S` flag | Not affected | | |
| `Z` flag | Not affected | | |
| `H` flag | Not affected | | |
| `P/V` flag | Not affected | | |
| `N` flag | Not affected | | |
| `C` flag | Not affected | | |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| res 0,a | $CB, $87 | 2F | X | X |
| res 0,b | $CB, $80 | 2F | X | X |
| res 0,c | $CB, $81 | 2F | X | X |
| res 0,d | $CB, $82 | 2F | X | X |
| res 0,e | $CB, $83 | 2F | X | X |
| res 0,h | $CB, $84 | 2F | X | X |
| res 0,l | $CB, $85 | 2F | X | X |
| res 0,(hl) | $CB, $86 | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| res 0,(ix+n) | $DD, $CB, $86, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| res 0,(iy+n) | $FD, $CB, $86, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| res 1,a | $CB, $8F | 2F | X | X |
| res 1,b | $CB, $88 | 2F | X | X |
| res 1,c | $CB, $89 | 2F | X | X |
| res 1,d | $CB, $8A | 2F | X | X |
| res 1,e | $CB, $8B | 2F | X | X |
| res 1,h | $CB, $8C | 2F | X | X |
| res 1,l | $CB, $8D | 2F | X | X |
| res 1,(hl) | $CB, $8E | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| res 1,(ix+n) | $DD, $CB, $8E, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| res 1,(iy+n) | $FD, $CB, $8E, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| res 2,a | $CB, $97 | 2F | X | X |
| res 2,b | $CB, $90 | 2F | X | X |
| res 2,c | $CB, $91 | 2F | X | X |
| res 2,d | $CB, $92 | 2F | X | X |
| res 2,e | $CB, $93 | 2F | X | X |
| res 2,h | $CB, $94 | 2F | X | X |
| res 2,l | $CB, $95 | 2F | X | X |
| res 2,(hl) | $CB, $96 | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| res 2,(ix+n) | $DD, $CB, $96, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| res 2,(iy+n) | $FD, $CB, $96, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| res 3,a | $CB, $9F | 2F | X | X |
| res 3,b | $CB, $98 | 2F | X | X |
| res 3,c | $CB, $99 | 2F | X | X |
| res 3,d | $CB, $9A | 2F | X | X |
| res 3,e | $CB, $9B | 2F | X | X |
| res 3,h | $CB, $9C | 2F | X | X |
| res 3,l | $CB, $9D | 2F | X | X |
| res 3,(hl) | $CB, $9E | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| res 3,(ix+n) | $DD, $CB, $9E, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| res 3,(iy+n) | $FD, $CB, $9E, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| res 4,a | $CB, $A7 | 2F | X | X |
| res 4,b | $CB, $A0 | 2F | X | X |
| res 4,c | $CB, $A1 | 2F | X | X |
| res 4,d | $CB, $A2 | 2F | X | X |
| res 4,e | $CB, $A3 | 2F | X | X |
| res 4,h | $CB, $A4 | 2F | X | X |
| res 4,l | $CB, $A5 | 2F | X | X |
| res 4,(hl) | $CB, $A6 | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| res 4,(ix+n) | $DD, $CB, $A6, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| res 4,(iy+n) | $FD, $CB, $A6, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |

Continued on next page

Table 9.2 – continued from previous page

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| res 5,a | $CB, $AF | 2F | X | X |
| res 5,b | $CB, $A8 | 2F | X | X |
| res 5,c | $CB, $A9 | 2F | X | X |
| res 5,d | $CB, $AA | 2F | X | X |
| res 5,e | $CB, $AB | 2F | X | X |
| res 5,h | $CB, $AC | 2F | X | X |
| res 5,l | $CB, $AD | 2F | X | X |
| res 5,(hl) | $CB, $AE | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| res 5,(ix+n) | $DD, $CB, $AE, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| res 5,(iy+n) | $FD, $CB, $AE, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| res 6,a | $CB, $B7 | 2F | X | X |
| res 6,b | $CB, $B0 | 2F | X | X |
| res 6,c | $CB, $B1 | 2F | X | X |
| res 6,d | $CB, $B2 | 2F | X | X |
| res 6,e | $CB, $B3 | 2F | X | X |
| res 6,h | $CB, $B4 | 2F | X | X |
| res 6,l | $CB, $B5 | 2F | X | X |
| res 6,(hl) | $CB, $B6 | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| res 6,(ix+n) | $DD, $CB, $B6, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| res 6,(iy+n) | $FD, $CB, $B6, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| res 7,a | $CB, $BF | 2F | X | X |
| res 7,b | $CB, $B8 | 2F | X | X |
| res 7,c | $CB, $B9 | 2F | X | X |
| res 7,d | $CB, $BA | 2F | X | X |
| res 7,e | $CB, $BB | 2F | X | X |
| res 7,h | $CB, $BC | 2F | X | X |
| res 7,l | $CB, $BD | 2F | X | X |
| res 7,(hl) | $CB, $BE | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| res 7,(ix+n) | $DD, $CB, $BE, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| res 7,(iy+n) | $FD, $CB, $BE, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |

**Notes**

- Interestingly enough, the index registers `IXH`, `IXL`, `IYH`, and `IYL` cannot be used in this instruction, yet `(IX+n)` and `(IY+n)` are allowed as operands.

**See Also** BIT, SET

## SET

**SET b,M** Set Bit

**Description**

Sets the `` b``th bit of ``M `` to one. `b` must be a hard-coded number between 0 and 7.

**Results**

| Register/Flag | 8-bit | 16-bit (non-ADL) | 24-bit (ADL) |
|---|---|---|---|
| `M` | The ‘‘b‘‘th bit of M is set to one | | |
| `S` flag | Not affected | | |
| `Z` flag | Not affected | | |
| `H` flag | Not affected | | |
| `P/V` flag | Not affected | | |
| `N` flag | Not affected | | |
| `C` flag | Not affected | | |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| set 0,a | $CB, $C7 | 2F | X | X |
| set 0,b | $CB, $C0 | 2F | X | X |
| set 0,c | $CB, $C1 | 2F | X | X |
| set 0,d | $CB, $C2 | 2F | X | X |
| set 0,e | $CB, $C3 | 2F | X | X |
| set 0,h | $CB, $C4 | 2F | X | X |
| set 0,l | $CB, $C5 | 2F | X | X |
| set 0,(hl) | $CB, $C6 | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| set 0,(ix+n) | $DD, $CB, $C6, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| set 0,(iy+n) | $FD, $CB, $C6, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| set 1,a | $CB, $CF | 2F | X | X |
| set 1,b | $CB, $C8 | 2F | X | X |
| set 1,c | $CB, $C9 | 2F | X | X |
| set 1,d | $CB, $CA | 2F | X | X |
| set 1,e | $CB, $CB | 2F | X | X |
| set 1,h | $CB, $CC | 2F | X | X |
| set 1,l | $CB, $CD | 2F | X | X |
| set 1,(hl) | $CB, $CE | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| set 1,(ix+n) | $DD, $CB, $CE, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| set 1,(iy+n) | $FD, $CB, $CE, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| set 2,a | $CB, $D7 | 2F | X | X |
| set 2,b | $CB, $D0 | 2F | X | X |
| set 2,c | $CB, $D1 | 2F | X | X |
| set 2,d | $CB, $D2 | 2F | X | X |
| set 2,e | $CB, $D3 | 2F | X | X |
| set 2,h | $CB, $D4 | 2F | X | X |
| set 2,l | $CB, $D5 | 2F | X | X |
| set 2,(hl) | $CB, $D6 | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| set 2,(ix+n) | $DD, $CB, $D6, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| set 2,(iy+n) | $FD, $CB, $D6, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| set 3,a | $CB, $DF | 2F | X | X |
| set 3,b | $CB, $D8 | 2F | X | X |
| set 3,c | $CB, $D9 | 2F | X | X |
| set 3,d | $CB, $DA | 2F | X | X |
| set 3,e | $CB, $DB | 2F | X | X |
| set 3,h | $CB, $DC | 2F | X | X |
| set 3,l | $CB, $DD | 2F | X | X |
| set 3,(hl) | $CB, $DE | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| set 3,(ix+n) | $DD, $CB, $DE, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| set 3,(iy+n) | $FD, $CB, $DE, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |

Continued on next page

Table 9.3 – continued from previous page

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| set 4,a | $CB, $E7 | 2F | X | X |
| set 4,b | $CB, $E0 | 2F | X | X |
| set 4,c | $CB, $E1 | 2F | X | X |
| set 4,d | $CB, $E2 | 2F | X | X |
| set 4,e | $CB, $E3 | 2F | X | X |
| set 4,h | $CB, $E4 | 2F | X | X |
| set 4,l | $CB, $E5 | 2F | X | X |
| set 4,(hl) | $CB, $E6 | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| set 4,(ix+n) | $DD, $CB, $E6, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| set 4,(iy+n) | $FD, $CB, $E6, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| set 5,a | $CB, $EF | 2F | X | X |
| set 5,b | $CB, $E8 | 2F | X | X |
| set 5,c | $CB, $E9 | 2F | X | X |
| set 5,d | $CB, $EA | 2F | X | X |
| set 5,e | $CB, $EB | 2F | X | X |
| set 5,h | $CB, $EC | 2F | X | X |
| set 5,l | $CB, $ED | 2F | X | X |
| set 5,(hl) | $CB, $EE | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| set 5,(ix+n) | $DD, $CB, $EE, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| set 5,(iy+n) | $FD, $CB, $EE, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| set 6,a | $CB, $F7 | 2F | X | X |
| set 6,b | $CB, $F0 | 2F | X | X |
| set 6,c | $CB, $F1 | 2F | X | X |
| set 6,d | $CB, $F2 | 2F | X | X |
| set 6,e | $CB, $F3 | 2F | X | X |
| set 6,h | $CB, $F4 | 2F | X | X |
| set 6,l | $CB, $F5 | 2F | X | X |
| set 6,(hl) | $CB, $F6 | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| set 6,(ix+n) | $DD, $CB, $F6, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| set 6,(iy+n) | $FD, $CB, $F6, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| set 7,a | $CB, $FF | 2F | X | X |
| set 7,b | $CB, $F8 | 2F | X | X |
| set 7,c | $CB, $F9 | 2F | X | X |
| set 7,d | $CB, $FA | 2F | X | X |
| set 7,e | $CB, $FB | 2F | X | X |
| set 7,h | $CB, $FC | 2F | X | X |
| set 7,l | $CB, $FD | 2F | X | X |
| set 7,(hl) | $CB, $FE | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| set 7,(ix+n) | $DD, $CB, $FE, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| set 7,(iy+n) | $FD, $CB, $FE, n | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |

**Notes**

- Interestingly enough, the index registers `IXH`, `IXL`, `IYH`, and `IYL` cannot be used in this instruction, yet `(IX+n)` and `(IY+n)` are allowed as operands.

**See Also** BIT, RES

CHAPTER 10

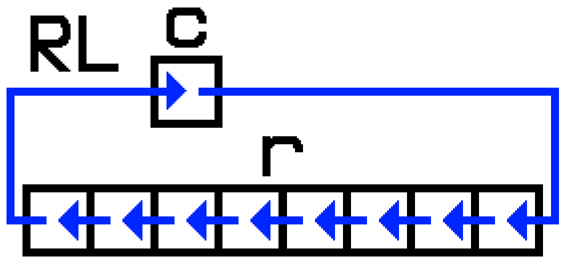
# Bit Shifts

## RL

**RL M** Rotate Left

**Description**

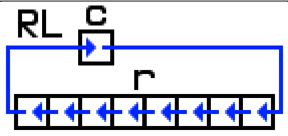
Performs a left shift on `M`; the 7th bit of `M` is moved into the carry, and the carry is moved into the 0th bit of `M`.



**Uses**

- Quickly multiplying by two
- Retrieving consecutive bits of a byte in a loop, starting from the left

**Results**

| Register/Flag | 8-bit |
|---|---|
| `M` |  |
| `S` flag | Set if the result is negative; else reset |
| `Z` flag | Set if the result is 0; else reset |
| `H` flag | Reset |
| `P/V` flag | Set if the result has even parity; else reset |
| `N` flag | Reset |
| `C` flag | 7th bit of `M` |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| rl a | $CB, $17 | 2F | X | X |
| rl b | $CB, $10 | 2F | X | X |
| rl c | $CB, $11 | 2F | X | X |
| rl d | $CB, $12 | 2F | X | X |
| rl e | $CB, $13 | 2F | X | X |
| rl h | $CB, $14 | 2F | X | X |
| rl l | $CB, $15 | 2F | X | X |
| rl (hl) | $CB, $16 | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| rl (ix+n) | $DD, $CB, n, $16 | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| rl (iy+n) | $FD, $CB, n, $16 | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |

**Notes**

- When using `RL` to multiply a number by 2, first make sure the carry flag is reset or the result will be `( M * 2 ) + Carry`.

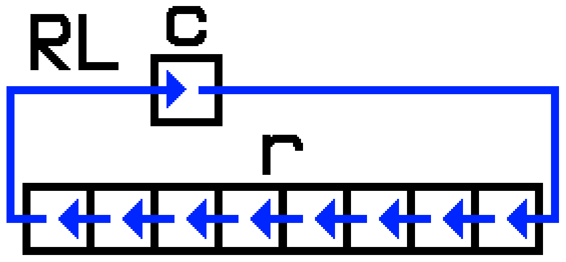
**See Also** RLA, RLC, RR, SLA

## RLA

**RLA** Rotate Left Accumulator

**Description**

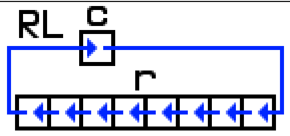
Performs a very fast `RL A`; the 7th bit of `A` is moved into the carry, and the carry is moved into the 0th bit of `A`.



**Uses**

- Quickly multiplying `A` by two
- Retrieving consecutive bits of a byte in a loop, starting from the left

**Results**

| Register/Flag | 8-bit |
|---|---|
| `A` |  |
| `S` flag | Not affected |
| `Z` flag | Not affected |
| `H` flag | Reset |
| `P/V` flag | Not affected |
| `N` flag | Reset |
| `C` flag | 7th bit of `A` |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) |
|---|---|---|
| rla | $17 | 1F |

**Notes**

- When using `RLA` to multiply `A` by 2, first make sure the carry flag is reset the result will be `( A * 2 ) + Carry`.
- Unlike `RL A`, this instruction does not meaningfully affect any flags except `C`.

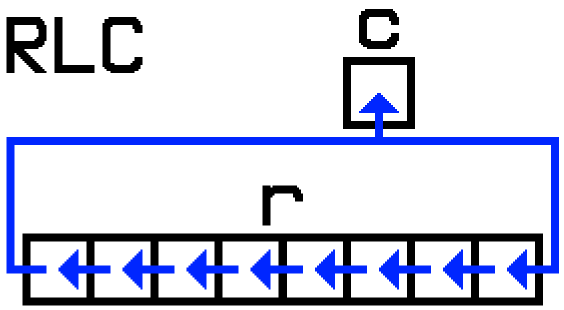
**See Also** RLA, RLC, RR, SLA

## RLC

**RLC M** Rotate Left Circular

**Description**

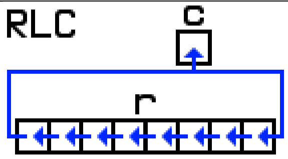
Performs a left shift on `M`; the 7th bit of `M` is copied into the carry and into the 0th bit of `M`.



**Uses**

- Retrieving consecutive bits of a byte in a loop, starting from the left, without affecting the source data (after eight iterations

**Results**

| Register/Flag | 8-bit |
|---|---|
| `M` |  |
| `S` flag | Set if the result is negative; else reset |
| `Z` flag | Set if the result is 0; else reset |
| `H` flag | Reset |
| `P/V` flag | Set if the result has even parity; else reset |
| `N` flag | Reset |
| `C` flag | 7th bit of `M` |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| rlc a | $CB, $07 | 2F | X | X |
| rlc b | $CB, $00 | 2F | X | X |
| rlc c | $CB, $01 | 2F | X | X |
| rlc d | $CB, $02 | 2F | X | X |
| rlc e | $CB, $03 | 2F | X | X |
| rlc h | $CB, $04 | 2F | X | X |
| rlc l | $CB, $05 | 2F | X | X |
| rlc (hl) | $CB, $06 | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| rlc (ix+n) | $DD, $CB, n, $06 | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| rlc (iy+n) | $FD, $CB, n, $06 | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |

**Notes**

- Unlike `RL`, the initial state of the carry flag has no effect on the result of this instruction.

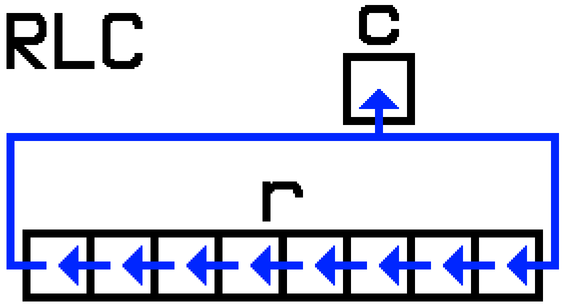
**See Also** RL, RLCA, RRC, SLA

## RLCA

**RLCA** Rotate Left Circular Accumulator

**Description**

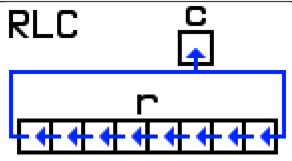
Performs a very fast `RLC A`; the 7th bit of `A` is copied into the carry and into the 0th bit of `A`.



**Uses**

- Retrieving consecutive bits of a byte in a loop, starting from the left, without affecting the source data (after eight iterations)

**Results**

| Register/Flag | 8-bit |
|---|---|
| `A` |  |
| `S` flag | Not affected |
| `Z` flag | Not affected |
| `H` flag | Reset |
| `P/V` flag | Not affected |
| `N` flag | Reset |
| `C` flag | 7th bit of `A` |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) |
|---|---|---|
| rlca | $07 | 1F |

**Notes**

- Unlike `RL`, the initial state of the carry flag has no effect on the result of this instruction.
- Unlike `RLC A`, this instruction does not meaningfully affect any flags except `C`.

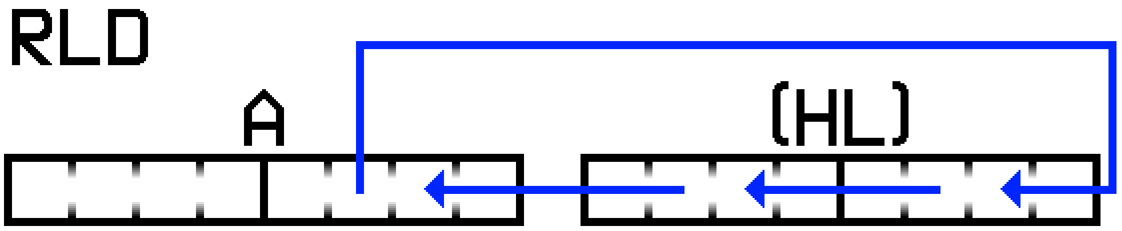
**See Also** RLA, RLC, RRCA

## RLD

**RLD** Rotate Left Decimal

**Description**

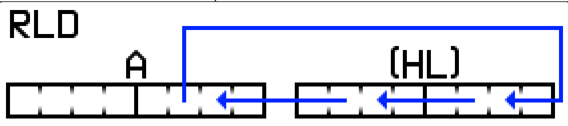
Moves the lower four bits of `HL` (0-3) into the upper four bits of `(HL)` (4-7); moves the upper four bits of `(HL)` (4-7) into the lower four bits of `A` (0-3); moves the lower four bits of `A` (0-3) into the lower four bits of `(HL)` (0-3).



**Uses**

- Shifting digits around with binary-coded decimal?

**Results**

| Register/Flag | 16-bit (non-ADL) | 24-bit (ADL) |
|---|---|---|
| `A` and `(HL)` |  | |
| `S` flag | Set if `A` is negative; else reset | |
| `Z` flag | Set if `A` is 0; else reset | |
| `H` flag | Reset | |
| `P/V` flag | Set if `A` has even parity; else reset | |
| `N` flag | Reset | |
| `C` flag | Not affected | |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| rrd | $ED, $6F | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |

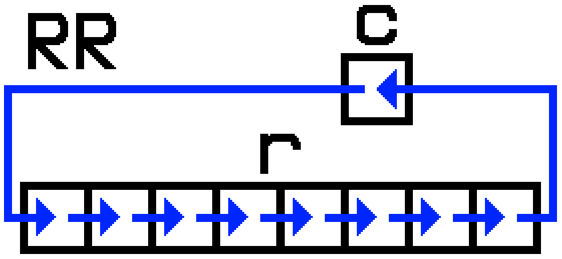
**See Also** DAA, RL, RLC, RRD

## RR

**RR M** Rotate Right

**Description**

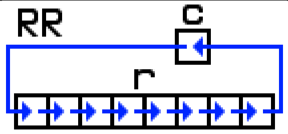
Performs a right shift on M; the 0th bit of M is moved into the carry, and the carry is moved into the 7th bit of M.



**Uses**

- Quickly dividing by two (carry flag is set if there was a remainder)
- Retrieving consecutive bits of a byte in a loop, starting from the right

**Results**

| Register/Flag | 8-bit |
|---|---|
| M |  |
| S flag | Set if the result is negative; else reset |
| Z flag | Set if the result is 0; else reset |
| H flag | Reset |
| P/V flag | Set if the result has even parity; else reset |
| N flag | Reset |
| C flag | 0th bit of M |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| rr a | $CB, $1F | 2F | X | X |
| rr b | $CB, $18 | 2F | X | X |
| rr c | $CB, $19 | 2F | X | X |
| rr d | $CB, $1A | 2F | X | X |
| rr e | $CB, $1B | 2F | X | X |
| rr h | $CB, $1C | 2F | X | X |
| rr l | $CB, $1D | 2F | X | X |
| rr (hl) | $CB, $1E | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| rr (ix+n) | $DD, $CB, n, $1E | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| rr (iy+n) | $FD, $CB, n, $1E | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |

**Notes**

- When using RR to divide a number by 2, first make sure the carry flag is reset or the result will be `( M * 2 ) + ( 128 * Carry )`.

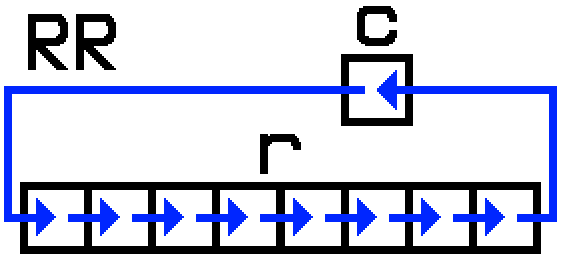
**See Also** RL, RRA, RRC, SRA, SRL

## RRA

**RRA** Rotate Right Accumulator

**Description**

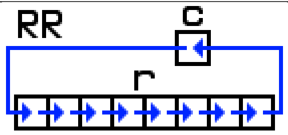
Performs a very fast RR A; the 0th bit of A is moved into the carry, and the carry is moved into the 7th bit of A.



**Uses**

- Quickly dividing `A` by two (carry flag is set if there was a remainder)
- Retrieving consecutive bits of a byte in a loop, starting from the right

**Results**

| Register/Flag | 8-bit |
|---|---|
| `A` |  |
| `S` flag | Not affected |
| `Z` flag | Not affected |
| `H` flag | Reset |
| `P/V` flag | Not affected |
| `N` flag | Reset |
| `C` flag | 0th bit of `A` |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) |
|---|---|---|
| rra | $1F | 1F |

**Notes**

- When using `RRA` to divide `A` by 2, first make sure the carry flag is reset or the result will be `( A * 2 ) + ( 128 * Carry )`.
- Unlike `RR A`, this instruction does not meaningfully affect any flags except `C`.

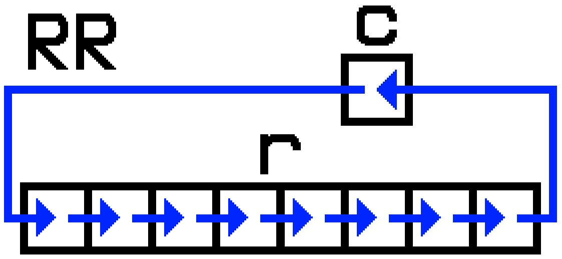
**See Also** RLA, RR, RRCA

## RRC

**RRC M** Rotate Right Circular

**Description**

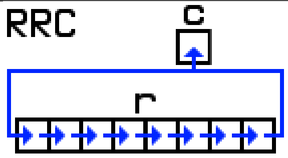
Performs a right shift on `M`; the 0th bit of `M` is copied into the carry and into the 7th bit of `M`.



**Uses**

- Retrieving consecutive bits of a byte in a loop, starting from the right, without affecting the source data (after eight iterations)

**Results**

| Register/Flag | 8-bit |
|---|---|
| M |  |
| S flag | Set if the result is negative; else reset |
| Z flag | Set if the result is 0; else reset |
| H flag | Reset |
| P/V flag | Set if the result has even parity; else reset |
| N flag | Reset |
| C flag | 0th bit of M |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| rr a | $CB, $0F | 2F | X | X |
| rr b | $CB, $08 | 2F | X | X |
| rr c | $CB, $09 | 2F | X | X |
| rr d | $CB, $0A | 2F | X | X |
| rr e | $CB, $0B | 2F | X | X |
| rr h | $CB, $0C | 2F | X | X |
| rr l | $CB, $0D | 2F | X | X |
| rr (hl) | $CB, $0E | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| rr (ix+n) | $DD, $CB, n, $0E | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| rr (iy+n) | $FD, $CB, n, $0E | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |

**Notes**

- Unlike RR, the initial state of the carry flag has no effect on the result of this instruction.

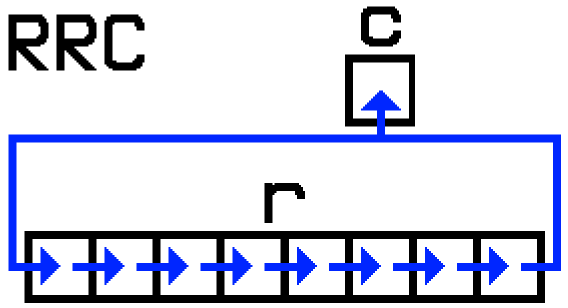
**See Also** RLC, RR, RRCA, SRA, SRL

## RRCA

**RRCA** Rotate Right Circular Accumulator

**Description**

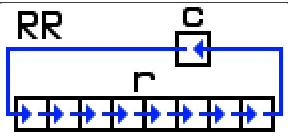
Performs a very fast RRC A; the 0th bit of A is copied into the carry and into the 7th bit of A.



**Uses**

- Retrieving consecutive bits of a byte in a loop, starting from the right, without affecting the source data (after eight iterations)

**Results**

| Register/Flag | 8-bit |
|---|---|
| A |  |
| S flag | Not affected |
| Z flag | Not affected |
| H flag | Reset |
| P/V flag | Not affected |
| N flag | Reset |
| C flag | 0th bit of A |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) |
|---|---|---|
| rra | $0F | 1F |

**Notes**

- Unlike RR, the initial state of the carry flag has no effect on the result of this instruction.
- Unlike RRC A, this instruction does not meaningfully affect any flags except C.

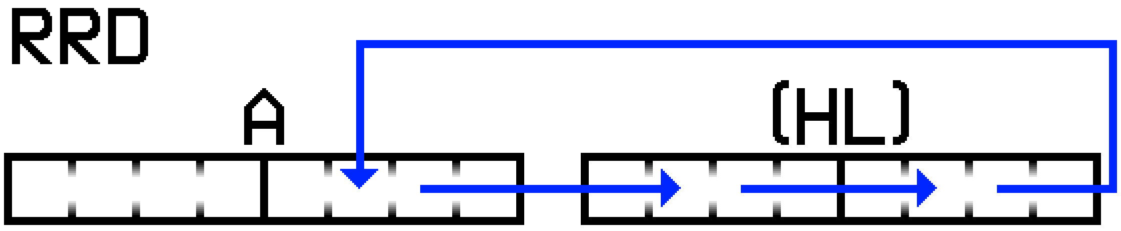
**See Also** RLCA, RRA, RRC

## RRD

**RRD** Rotate Right Decimal

**Description**

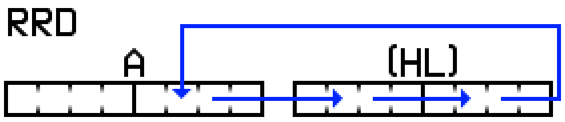
Moves the lower four bits of A (0-3) into the upper four bits of (HL) (4-7); moves the upper four bits of (HL) (4-7) into the lower four bits of (HL) (0-3); moves the lower four bits of (HL) (0-3) into the lower four bits of A (0-3).



**Uses**

- Shifting digits around with binary-coded decimal?

**Results**

| Register/Flag | 16-bit (non-ADL) | 24-bit (ADL) |
|---|---|---|
| `A` and `(HL)` |  | |
| `S` flag | Set if `A` is negative; else reset | |
| `Z` flag | Set if `A` is 0; else reset | |
| `H` flag | Reset | |
| `P/V` flag | Set if `A` has even parity; else reset | |
| `N` flag | Reset | |
| `C` flag | Not affected | |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| rrd | $ED, $67 | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |

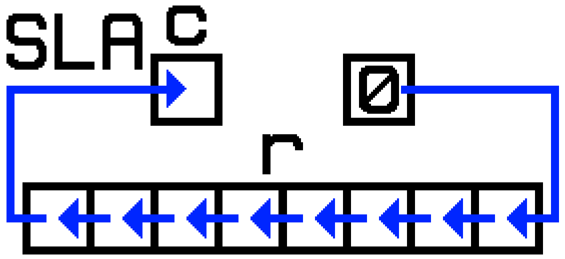
**See Also** DAA, RLD, RR, RRC

## SLA

**SLA M** Shift Left Arithmetic

**Description**

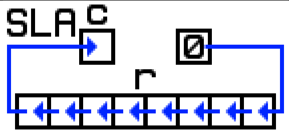
Performs a left shift on `M`; the 7th bit of `M` is moved into the carry, and a zero is placed in the 0th bit of `M`.



**Uses**

- Quickly multiplying by two

**Results**

| Register/Flag | 8-bit |
|---|---|
| `M` |  |
| `S` flag | Set if the result is negative; else reset |
| `Z` flag | Set if the result is 0; else reset |
| `H` flag | Reset |
| `P/V` flag | Set if the result has even parity; else reset |
| `N` flag | Reset |
| `C` flag | 7th bit of `M` |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| sla a | $CB, $27 | 2F | X | X |
| sla b | $CB, $20 | 2F | X | X |
| sla c | $CB, $21 | 2F | X | X |
| sla d | $CB, $22 | 2F | X | X |
| sla e | $CB, $23 | 2F | X | X |
| sla h | $CB, $24 | 2F | X | X |
| sla l | $CB, $25 | 2F | X | X |
| sla (hl) | $CB, $26 | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| sla (ix+n) | $DD, $CB, n, $26 | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| sla (iy+n) | $FD, $CB, n, $26 | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |

**Notes**

- Unlike `RL`, the initial state of the carry flag has no effect on the result of this instruction, so a number can be easily doubled without worrying about the state of the carry flag.
- Despite the mnemonics, `SLA` has more in common with `SRL` (Shift Right Logical) than `SRA` (Shift Right Arithmetic). This is because `SRA` is intended for use with signed integers, whereas `SLA` and `SRL` are more useful with unsigned integers.

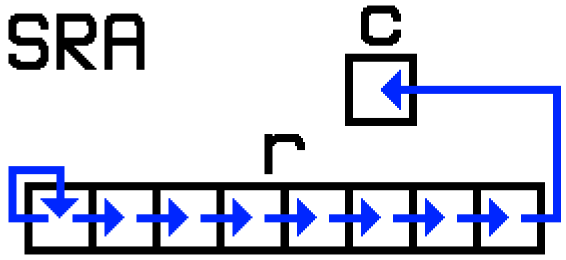
**See Also** RL, RLC, SRA, SRL

## SRA

**SRA M** Shift Right Arithmetic

**Description**

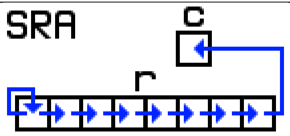
Performs a right shift on `M`; the 7th bit of `M` remains unchanged, and the 0th bit of `M` is moved into the carry.



**Uses**

- Quickly dividing a signed integer by two

**Results**

| Register/Flag | 8-bit |
|---|---|
| `M` |  |
| `S` flag | Set if the result is negative; else reset |
| `Z` flag | Set if the result is 0; else reset |
| `H` flag | Reset |
| `P/V` flag | Set if the result has even parity; else reset |
| `N` flag | Reset |
| `C` flag | 0th bit of `M` |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| sra a | $CB, $2F | 2F | X | X |
| sra b | $CB, $28 | 2F | X | X |
| sra c | $CB, $29 | 2F | X | X |
| sra d | $CB, $2A | 2F | X | X |
| sra e | $CB, $2B | 2F | X | X |
| sra h | $CB, $2C | 2F | X | X |
| sra l | $CB, $2D | 2F | X | X |
| sra (hl) | $CB, $2E | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| sra (ix+n) | $DD, $CB, n, $2E | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| sra (iy+n) | $FD, $CB, n, $2E | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |

**Notes**

- Unlike `RR`, the initial state of the carry flag has no effect on the result of this instruction, so a signed number can be easily halved without worrying about the state of the carry flag.
- Despite the mnemonics, `SRA` is not very similar to `SLA` (Shift Left Arithmetic), which is meant for unsigned integers. For a right-shifting version of `SLA`, use `SRL` (Shift Right Logical).

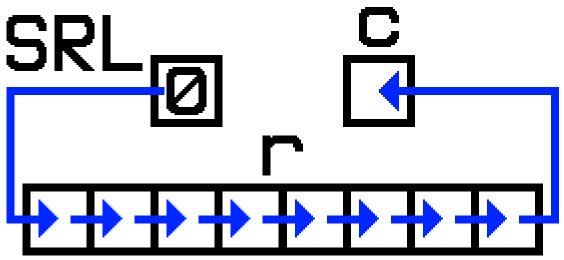
**See Also** RR, RRC, SLA, SRL

## SRL

**SRL M** Shift Right Logical

**Description**

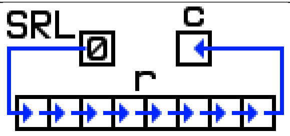
Performs a right shift on `M`; the 0th bit of `M` is moved into the carry, and a zero is placed in the 7th bit of `M`.



**Uses**

- Quickly dividing a number by two

**Results**

| Register/Flag | 8-bit |
|---|---|
| `M` |  |
| `S` flag | Set if the result is negative; else reset |
| `Z` flag | Set if the result is 0; else reset |
| `H` flag | Reset |
| `P/V` flag | Set if the result has even parity; else reset |
| `N` flag | Reset |
| `C` flag | 0th bit of `M` |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| srl a | $CB, $3F | 2F | X | X |
| srl b | $CB, $38 | 2F | X | X |
| srl c | $CB, $39 | 2F | X | X |
| srl d | $CB, $3A | 2F | X | X |
| srl e | $CB, $3B | 2F | X | X |
| srl h | $CB, $3C | 2F | X | X |
| srl l | $CB, $3D | 2F | X | X |
| srl (hl) | $CB, $3E | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |
| srl (ix+n) | $DD, $CB, n, $3E | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |
| srl (iy+n) | $FD, $CB, n, $3E | 4F + 1R + 1W + 1 | 5F + 1R + 1W + 1 | 5F + 1R + 1W + 1 |

**Notes**

- Unlike `RR`, the initial state of the carry flag has no effect on the result of this instruction, so a number can be easily halved without worrying about the state of the carry flag.
- Despite the mnemonics, the reverse of `SRL` is actually `SLA` (Shift Left Arithmetic).

**See Also** RL, RLC, SRA, SRL

CHAPTER 11

# Block Operations

## CPD

**CPD** Compare and Decrement

**Description**

Performs `cp a,(hl)`, then decrements `HL` and `BC`.

```
cp a,(hl)
dec hl
dec bc
```

**Uses**

- This instruction is not very useful on its own, as `cp a,(hl)` is both smaller and faster than a single `CPD`, but more often used is the repeating version of `CPD`: `CPDR`

**Results**

| Register/Flag | 16-bit (non-ADL) | 24-bit (ADL) |
|---|---|---|
| `S` flag | Set if result is negative; else reset | |
| `Z` flag | Set if `A` = `(HL)`; else reset | |
| `H` flag | Set if borrow from bit 4; else reset | |
| `P/V` flag | Set if `BC`  0 after the operation; else reset | |
| `N` flag | Set | |
| `C` flag | Not affected | |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| cpd | \$ED, \$A9 | 2F + 1R | 3F + 1R | 3F + 1R |

**See Also** CP, CPDR, CPI, CPIR, LDD

## CPDR

**CPDR** Compare and Decrement with Repeat

**Description**

Performs `cpd` until either `A` = `(HL)` or `BC` = 0.

```
cp a,(hl)
dec hl
dec bc
jr nz,-7
ret po
jr -10
```

**Uses**

- Finding a certain letter in a string (or other similar tasks)

**Results**

| Register/Flag | 16-bit (non-ADL) | 24-bit (ADL) |
|---|---|---|
| `S` flag | Set if result is negative; else reset | |
| `Z` flag | Set if `A` = `(HL)`; else reset | |
| `H` flag | Set if borrow from bit 4; else reset | |
| `P/V` flag | Set if `BC`  0 after the operation; else reset | |
| `N` flag | Set | |
| `C` flag | Not affected | |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| cpir | $ED, $B1 | `2F + (Iterations)*(1R + 2) - 1` where `Iterations` is the number of iterations of `cpd` before either `A` = `(HL)` or `BC` = 0 | 3F + (Iterations)*(1R + 2) - 1 | 3F + (Iterations)*(1R + 2) - 1 |

**Notes**

- Interrupts can be triggered while this instruction is in progress (unless they are disabled using `DI`, of course).

**See Also** CP, CPD, CPI, CPIR, LDDR

## CPI

**CPI** Compare and Increment

**Description**

Performs `cp a,(hl)`, then increments `HL` and decrements `BC`.

```
cp a,(hl)
inc hl
dec bc
```

**Uses**

- This instruction is not very useful on its own, as `cp a,(hl)` is both smaller and faster than a single `CPI`, but more often used is the repeating version of `CPI`: `CPIR`

**Results**

| Register/Flag | 16-bit (non-ADL) | 24-bit (ADL) |
|---|---|---|
| `S` flag | Set if result is negative; else reset | |
| `Z` flag | Set if `A` = `(HL)`; else reset | |
| `H` flag | Set if borrow from bit 4; else reset | |
| `P/V` flag | Set if `BC`  0 after the operation; else reset | |
| `N` flag | Set | |
| `C` flag | Not affected | |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| cpi | $ED, $A1 | 2F + 1R | 3F + 1R | 3F + 1R |

**Notes**

- Although this instruction increments `HL`, it decrements `BC`.

**See Also** CP, CPD, CPDR, CPIR, LDI

## CPIR

**CPIR** Compare and Increment with Repeat

**Description**

Performs `cpi` until either `A` = `(HL)` or `BC` = 0.

```
cp a,(hl)
inc hl
dec bc
jr nz,-7
ret po
jr -10
```

**Uses**

- Finding a certain letter in a string (or other similar tasks)

**Results**

| Register/Flag | 16-bit (non-ADL) | 24-bit (ADL) |
|---|---|---|
| `S` flag | Set if result is negative; else reset | |
| `Z` flag | Set if `A` = `(HL)`; else reset | |
| `H` flag | Set if borrow from bit 4; else reset | |
| `P/V` flag | Set if `BC`  0 after the operation; else reset | |
| `N` flag | Set | |
| `C` flag | Not affected | |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| cpir | $ED, $B9 | `2F + (Iterations)*(1R + 2) - 1` where `Iterations` is the number of iterations of `cpi` before either `A` = `(HL)` or `BC` = 0 | 3F + (Iterations)*(1R + 2) - 1 | 3F + (Iterations)*(1R + 2) - 1 |

**Notes**

- Interrupts can be triggered while this instruction is in progress (unless they are disabled using `DI`, of course).

**See Also** CP, CPD, CPDR, CPI, LDIR

## LDD

**LDD** Load and Decrement

**Description**

Copies one byte from `(HL)` to `(DE)`, then decrements `HL`, `DE`, and `BC`.

```
ld (de),(hl) ; Not normally a valid instruction
dec hl
dec de
dec bc
```

**Uses**

- This instruction is not very useful on its own, but more often used is the repeating version of `LDD`: `LDDR`

**Results**

| Register/Flag | 16-bit (non-ADL) | 24-bit (ADL) |
|---|---|---|
| `S` flag | Not affected | |
| `Z` flag | Not affected | |
| `H` flag | Reset | |
| `P/V` flag | Set if `BC` 0 after the operation; else reset | |
| `N` flag | Reset | |
| `C` flag | Not affected | |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| ldi | $ED, $A8 | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |

**Notes**

- If all you want to accomplish is `ld (de),(hl)`, consider `ld a,(hl) \ ld (de),a`. It is the same size but is one clock cycle less and doesn't modify any of the registers except `A`.

**See Also** CPD, LD, LDI, LDDR, LDIR

## LDDR

**LDDR** Load and Decrement with Repeat

**Description**

Performs `ldd` until `BC` = 0, effectively copying `BC` bytes of data from `HL` to `DE`, where `HL` and `DE` point to the end of their respective blocks.

```
ldd
ret po
jr -5
```

**Uses**

- **Copying lots of data**
  - Sprite routines
  - Copying graphical data from some buffer to the actual screen
  - Filling lots of space with one (or more) bytes

**Results**

| Register/Flag | 16-bit (non-ADL) | 24-bit (ADL) |
|---|---|---|
| `S` flag | Not affected | |
| `Z` flag | Not affected | |
| `H` flag | Reset | |
| `P/V` flag | Set if `BC`  0 after the operation; else reset | |
| `N` flag | Reset | |
| `C` flag | Not affected | |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| lddr | $ED, $B8 | 2F + (1R + 1W + 1)* `BC` | 3F + (1R + 1W + 1)* `BC` | 3F + (1R + 1W + 1)* `BC` |

**Notes**

- Interrupts can be triggered while this instruction is in progress (unless they are disabled using `DI`, of course).
- Since most of the time it's more convenient to specify the beginning of a block of data than the end, `LDIR` is significantly more popular than `LDDR`.
- Assuming you are copying hard-coded data and BC is NOT already set to the desired number of bytes, it is only faster to use hard-coded `LDD` s instead of `ld bc,BytesToCopy \ lddr` if you are copying two bytes. Copying three bytes, the cycle times and size of the code are exactly the same (6 bytes, 6F+3R+3W+3).
- If you want to copy a few more bytes than whatever number is in `BC`, it is both smaller and significantly faster to use `INC BC` several times before `LDDR` than a few `LDD` s after it. (`INC BC` is one byte and only 1F (plus the extra 1R + 1W + 1 from `LDDR`) whereas `LDD` is two bytes and 2F+1R+1W+1. So you just save one byte and 1F.)

**Examples**

- Filling a block of memory with a single byte

```
ld hl,EndOfBlock
ld de,EndOfBlock-1
ld bc,SizeOfBlock-1
ld (hl),ByteToCopy
lddr
```

**See Also** CPDR, LD, LDD, LDI, LDIR

## LDI

**LDI** Load and Increment

**Description**

Copies one byte from `(HL)` to `(DE)`, then increments `HL`, increments `DE`, and decrements `BC`.

```
ld (de),(hl) ; Not normally a valid instruction
inc hl
inc de
dec bc
```

**Uses**

- This instruction is not very useful on its own, but more often used is the repeating version of `LDI`: `LDIR`

**Results**

| Register/Flag | 16-bit (non-ADL) | 24-bit (ADL) |
|---|---|---|
| `S` flag | Not affected | |
| `Z` flag | Not affected | |
| `H` flag | Reset | |
| `P/V` flag | Set if `BC`  0 after the operation; else reset | |
| `N` flag | Reset | |
| `C` flag | Not affected | |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| ldi | $ED, $A0 | 2F + 1R + 1W + 1 | 3F + 1R + 1W + 1 | 3F + 1R + 1W + 1 |

**Notes**

- Although this instruction increments `HL` and `DE`, it decrements `BC`.
- If all you want to accomplish is `ld (de),(hl)`, consider `ld a,(hl) \ ld (de),a`. It is the same size but is one clock cycle less and doesn't modify any of the registers except `A`.

**See Also** CPI, LD, LDD, LDDR, LDIR

## LDIR

**The Legendary LDIR** Load And Increment with Style

**Description**

Performs `ldi` until `BC` = 0, effectively copying `BC` bytes of data from `HL` to `DE`, where `HL` and `DE` point to the start of their respective blocks.

```
ldi
ret po
jr -5
```

**Uses**

- **Copying lots of data**
  - Sprite routines
  - Copying graphical data from some buffer to the actual screen
  - Filling lots of space with one (or more) bytes

**Results**

| Register/Flag | 16-bit (non-ADL) | 24-bit (ADL) |
|---|---|---|
| `S` flag | Not affected | |
| `Z` flag | Not affected | |
| `H` flag | Reset | |
| `P/V` flag | Set if `BC`  0 after the operation; else reset | |
| `N` flag | Reset | |
| `C` flag | Not affected | |

**Allowed Instructions**

| Instruction | Opcode | CC (ADL/non-ADL) | CC (.S) | CC (.L) |
|---|---|---|---|---|
| ldir | $ED, $B0 | 2F + (1R + 1W + 1)* `BC` | 3F + (1R + 1W + 1)* `BC` | 3F + (1R + 1W + 1)* `BC` |

**Notes**

- Interrupts can be triggered while this instruction is in progress (unless they are disabled using `DI`, of course).

- Assuming you are copying hard-coded data and BC is NOT already set to the desired number of bytes, it is only faster to use hard-coded `LDI` s instead of `ld bc,BytesToCopy \ ldir` if you are copying two bytes. Copying three bytes, the cycle times and size of the code are exactly the same (6 bytes, 6F+3R+3W+3).
- If you want to copy a few more bytes than whatever number is in `BC`, it is both smaller and significantly faster to use `INC BC` several times before `LDIR` than a few `LDI` s after it. (`INC BC` is one byte and only 1F (plus the extra 1R + 1W + 1 from `LDIR`) whereas `LDI` is two bytes and 2F+1R+1W+1. So you just save one byte and 1F.)

**Examples**

- Filling a block of memory with a single byte

```
ld hl,StartOfData
ld de,StartOfData+1
ld bc,SizeOfData-1
ld (hl),ByteToCopy
ldir
```

- Copy graphical data from vBuf2 to vBuf1

```
ld hl,vBuf2
ld de,vBuf1
ld bc,320*240
ldir
```

- Set up a hard-coded 8bpp palette

```
ld hl,Palette_Start
ld de,mpLcdPalette
ld bc,Palette_End-Palette_Start
ldir
```

- A simple 8bpp rectangle drawing routine (credit: MateoConLechuga)

```
; INPUTS
; BC                              Width
; DE                              X-position
; H                                       Height
; L                                       Y-position
; (FillRect_Color)        Color
FillRect:
        ld a,h ; Store the height in A to be used as a loop counter
        ld h,160
        mlt hl
        add hl,hl ; HL now contains the Y position multiplied by 320
        add hl,de ; Add in the X position...
        ld de,vBuf1
        add hl,de ; And the LCD memory location...
        ; Now HL is pointing to the first pixel of the rectangle
        dec bc ; Get the rectangle width minus 1 in BC (more on that␣
↪later)
FillRect_Loop:
FillRect_Color = $+1
        ld (hl),0 ; This is self-modifying code
        push hl
        pop de
        inc de ; Now DE = HL + 1
        push bc ; Save BC for later
        ldir ; Copy BC (width-1) bytes from HL (first pixel of this␣
↪row of the rectangle) to DE (next pixel)
        ; Now one row of the rectangle is done
        pop bc ; Grab BC again
        ld de,320
        add hl,de ; Advanced HL one pixel down...
        sbc hl,bc ; And return to the left edge of the rectangle
        dec a ; Decrement our loop counter...
        jr nz,FillRect_Loop ; And repeat if we haven't finished
        ret
```

**See Also** CPIR, LD, LDD, LDDR, LDI

# CHAPTER 12

# IO

# CHAPTER 13

# Load And Exchange

# CHAPTER 14

# Logic

# CHAPTER 15

# Program Flow

# CHAPTER 16

# Processor Control

# CHAPTER 17

# Hexcodes

## What are hex codes?

Hex codes are little assembly programs that TI-BASIC programmers can use to add functionality to their programs. Here is an example of how to use a hexcode.

```
Asm84CEPrgm
3A8705D0CD080B02CD300F02C9
```

Each hex code will have the program and the source in assembly, along with a brief description of the functionality.

**Make sure you have the FULL, correct hexcode. These programs can clear your ram if not entered correctly!**

## General

### Toggle Program Mode

When used in a program, it allows you to use Archive and UnArchive on other programs. Make sure to switch back to "program mode" when you're done by running the program again. When used on the home screen, it allows you to use programming commands like If and For(; this has limited utility, but maybe it's useful to check a short bit of code without creating a new program for it.

```
FD7E08EE02FD7708C9
```

```
ld a,(iy+8)
xor 2
ld (iy+8),a
ret
```

### Quick Key

This is a getKey routine that makes all keys repeat, not just arrows and there is no delay between repeats. The key codes are different, so you might need to experiment.

```
3A8705D0CD080B02CD300F02C9
```

```
ld a,(kbdScanCode)
call _SetXXOP1
call _StoAns
ret
```

### Text Inverse

This will switch from normal text mode to inverse (white text on black background) and vice versa.

```
FD7E05EE08FD7705C9
```

```
ld a,(iy+textFlags)
xor 1&lt;&lt;textInverse
ld (iy+textFlags),a
ret
```

## LCD

### LCD Clear

This only clears the LCD, it doesn't actually clear the graph screen or homescreen

```
CD080802C9
```

```
call _ClrLCDFull
ret
```

### Fill Color

This fills the LCD with pixels of the color you specify. Replace XX with the palettized color, list of available colors here: http://ce-programming.github.io/documentation/images/tutorials/asm/rgbhlpalette.png.

```
3EXX210000D401005802CDE01002C9
```

```
ld a,$XX
ld hl,vRam
ld bc,320*240*2
call _MemSet
ret
```

### Black

This fills the LCD with black pixels

```
3E00210000D401005802CDE01002C9
```

```
ld a,$00
ld hl,vRam
ld bc,320*240*2
call _MemSet
ret
```

### White

This fills the LCD with white pixels

```
3EFF210000D401005802CDE01002C9
```

```
ld a,$FF
ld hl,vRam
ld bc,320*240*2
call _MemSet
ret
```

## Run Indicator

### On

This turns on the run indicator.

```
CD440802C9
```

```
call _RunIndicOn
ret
```

### Off

This turns off the run indicator.

```
CD480802C9
```

```
call _RunIndicOff
ret
```

Hexcode descriptions originally from TI-BD.

# CHAPTER 18

# Resources

## More Websites

CE Programming Documentation: TI 84+CE Assembly and C Tutorials
Learn Assembly In 28 Days: TI 83+ Assembly Tutorial
Cemetech: Forums and Archives
WikiTI: 84+CE Technical Info

## Notepad++

Download Here: Notepad++
Syntax Highlighting: Notepad++ eZ80 Syntax Highlighting

## SPASM-ng

Download Here: SPASM-ng

## TI Connect CE

Download Here: TI Connect CE

## TI 84+CE Equates File

Download Here: ti84pce.inc

## TI 83+ System Routines PDF

Download here