Using SHORT (Two-byte) *Relative* Jump Instructions

[This page began as a reply to a question asked by Adam Drayer.]

Here we discuss the use of ***two-byte*** **JMP** instructions in x86 Assembly code.
Though we mention only JMP code, what you'll learn here about ***Relative*** **offsets** will also apply to *all* **Conditional Jumps** (such as JE, JG, JC, JZ, JNE, JNG, JNC, JNZ, etc.) as well!

These are also known as **SHORT *Relative* Jump**s. Programs using only Relative Jump instructions can be *relocated* anywhere in memory without having to change the machine code for the Jumps. The first byte of a SHORT Jump is always **EB** and the second is a *relative offset* from **00**h to **7F**h for **Forward** jumps, and from **80**h to **FF**h for **Reverse** (or Backward) jumps. **[Note:** The offset ***count*** always begins at the byte immediately ***after*** the **JMP** instruction for ***any*** type of Relative Jump!**]**

Whether you use a *label* to point to the next instruction or a specific *address* (as required by MS-**Debug**'s **A**ssemble command), all *Assemblers* still figure out the value of the offset byte for you. If you point to an *address* that's too far away for a SHORT Jump to reach, the *Assembler* should code the instruction as a ***three-byte*** **NEAR** Jump instead* (an *Absolute* **FAR** Jump is one that will jump outside of the present **64** KiB **C**ode **S**egment). Therefore, programmers who are trying to keep a routine down to the least number of bytes, must know the ***limits*** of both **Forward** and **Reverse** **SHORT** (and NEAR) Jumps!

---

*Note: MS-DEBUG's **A**ssembler will use the smallest possible JMP code (first SHORT, then NEAR and finally FAR) for any address you give it. The main reason it can do this is because the exact location of the next instruction must be specified. Most *Assemblers*, however, will create *space* for at least a 3-byte **NEAR** Jump even though it might not be necessary; *unless* you include a "**SHORT**" directive before the "JMP" mnemonic in your source code! **This may explain why you see a "NoOp" (90h) byte after a SHORT Jump in code that doesn't need an extra byte. With only a label name, Assemblers need more than one pass through your source code to know how far away (from a Jump instruction) that label name actually points to.** If you use a SHORT directive in your source and the address ends up being too far away for a SHORT Jump, you'll get an error message.

Forward SHORT Jumps

Forward Jumps are the easiest of the two to work with. They use relative offset values from **00**h to **7F**h which enable program execution to jump to another instruction with a ***maximum*** of **127** bytes ***in-between*** them. A jump *of any kind* to an instruction immediately following the code, would be just plain illogical; *unless* you wanted to *reserve* two or more bytes there in a *tricky manner* (*without* using **NOP**s; 90h bytes). There may, however, be some cases where one wishes to jump over a single-byte instruction.

The relative offset byte is essentially an 8-bit ***signed*** number where the most significant **bit** is **0** for *positive* numbers. Therefore, all the bytes from **zero** through **7F**h (**0**111 1111 binary) are ***positive*** and give us a **Forward** Jump.

For **JMP** instructions beginning at Offset **100**h, the following is true:

Second Byte Value	Bytes in-between	NextInstruction Location (Hex)
00	0	102
01	1	103
02	2	104
03	3	105
04	4	106
...	...	...
7c	124	17e
7d	125	17f
7e	126	180
7f	127	181

Formula:

JMP_Address **+ 2 +** Second_Byte_value **=** Next_Instruction_Address

Examples:

Address	Code	Instruction	Formula Examples
0100	EB 03	JMP 0105	100h + 2 + 03h = 105h
0152	EB 23	JMP 0177	152h + 2 + 23h = 177h
0173	EB 47	JMP 01BC	173h + 2 + 47h = 1BCh
0200	EB 7F	JMP 0281	200h + 2 + 7Fh = 281h

Reverse (or Backward) SHORT Jumps

Reverse (or **Backward**) **Jumps** have relative offset bytes from **80**h to **FF**h. Unlike Forward Jumps, the *seemingly largest* offset byte here actually indicates the ***shortest*** **backward** jump, *because* we must use the ***2's Complement**** of each ***negatively*** **signed** offset byte**!** Let's compute the 2's Complement for both the upper and lower limits of a SHORT ***Backward*** Jump:

First, you ***invert*** each **bit** of the offset byte (giving its **1's Complement**):
FFh (1111 1111) —> **00**h (**0**000 0000) and
80h (**1**000 0000) —> **7F**h (**0**111 1111).
Following this, you simply **add 1** to each intermediate value, then make it a negative number. So, the **2's Complement** of each byte is in reality:
FFh —> **-01**h (a -1) and
80h —> **-80**h (a -128); these are not only *conceptually* **negative** numbers, but also *electronically,* or there couldn't be ***backward*** jumps (the CPU *knows* they are negative offsets because the first byte **EB**, tells it this is a SHORT Jump instruction where any value from 80h to FFh is treated as such).

---

*I'll try not to get *too* technical here about the **mathematics**, but I must say this: If *only* simple 8-bit ***signed*** numbers were used, then **00**h would give us a *+0* (*positive* zero; though strictly speaking, zero is neither positive nor negative), ***but*** an **80**h (being **1**000 0000 **b**inary) would give us a *−0* (***negative* zero**)**!** So, one very good reason for using **2's Complement arithmetic** is to avoid having ***two*** *different* zeros**!**

Let's work through one more example in detail: If we have an offset byte of **AA**h, that would be 10101010 in Binary. So, it's **1's complement** would be:

0**1010101** or **55**h. Therefore, we'd have a **2's complement** of: **-56**h (-86). *But* how do we translate that into a real *backwards* jump? Well, assuming that the address of our SHORT JMP code is 0**696**h, we must first ***add*** **2** to get to the address of the instruction immediately following our JMP; 0**698**h being that location. And finally add our ***negative*** *2's complement value* of **-**56h to arrive at the next instruction address of: 0**642**h. (In MS-DEBUG, if you **E**nter the bytes **EB AA** at **696** [-e 696 eb aa], a *disassembly* of that address [-u 696 698] will show up as: **JMP 0642** on the screen.)

Since all Jump *counts* must begin with the byte ***after*** the JMP code, Reverse Jumps must count backwards *through their own code!* This means that a Reverse Jump must waste **2 offset value *counts*** in order to jump over itself *before* getting back to just the last byte of any instruction preceding it. Practically speaking, this means you should *never* see JMP code with an offset byte of **FF**h (which ends up inside the JMP instruction itself), *unless* the programmer had something rather *"clever"* in mind. Most businesses wouldn't consider such code as being *professional* though. One possible use of the code **EB FE** would be to 'lockup' program execution by putting it into an ***endless loop***; it would keep repeating the same Jump **to itself** over and over again**!** There are few, *if any,* practical jumps to the preceding two bytes, because we'd need at least one 2-byte instruction (such as "JZ *elsewhere*") to break out of the **loop** such code would form!

The furthest ***back*** that a **SHORT *Relative* JMP** can *reach* is to the first byte of any instruction with **127** bytes ***in-between*** it *and* whatever instruction is immediately ***after*** the **JMP** code. You can see by this, that ***both*** Reverse *and* Forward Jumps have the **same** *numerical reach,* ***but*** a Reverse Jump must *count back* through its own code first! So in reality, Reverse Jumps have only a ***maximum*** of**125** bytes ***between*** them and the first byte of the **Next** Instruction.

For **JMP** instructions beginning at Offset **200**h, the following is true:

Second Byte Value	**Logical NOT** (hex)	**Two's** (2's) Complement*	**Next Instruction Location (Hex)**	**Bytes in-between**
FF	00	-1	201	Possibly a clever trick, but very **Un-**Professional
FE	01	-2	200	**Endless Loop**
FD	02	-3	1FF	0
FC	03	-4	1FE	1
FB	04	-5	1FD	2

...	...	...	...	...
83	7C	-125	185	122
82	7D	-126	184	123
81	7E	-127	183	124
80	7F	-128	182	125
*Showing the fact that the 8-bit **signed** bytes here are all ***negative***.				

Formula:

JMP_Address **+ 2 + (**2's Complement of Second Byte value**) =** Next_Instruction_Address

Examples:

Address	Code	Instruction	Two's (2's) Complement	Formula Examples
0147	EB FC	JMP 0145	-4	**147h + 2 + (- 4) = 145h**
0152	EB D7	JMP 012B	-29h (-41)	**152h + 2 + (- 29h) = 12Bh**
0173	EB AF	JMP 0124	-51h (-81)	**173h + 2 + (- 51h) = 124h**
0200	EB 80	JMP 0182	-80h (-128)	**200h + 2 + (- 80h) = 182h**

And from our page about the Win98 **MBR**:

Address	Code	Instruction	Two's (2's) Complement	Formula Examples
06BF	EB 8A	JMP 064B	-76h (-119)	**6BFh + 2 + (- 76h) = 64Bh**
06BF	EB 80	JMP 0641	-80h (-128)	**6BFh + 2 + (- 80h) = 641h**

One can see that the farthest we could jump ***back*** to (**0641**) would be in the middle of the instruction at **0640**; *missing* the desired location (**063F**) by only two bytes! Therefore, the program uses the convenient JMP code at **064B** instead to accomplish the task.

For those who are still somewhat confused, here's an example program that's little more than a whole bunch of Forward and Reverse JMP instructions for you to examine under a debugger! But, **JMP.COM** (inside **JMP.zip**) is a real program. Upon execution, it will display: "**Our JMP tour has ended... Goodbye!**"

Updated: October 14, 2004 (2004.10.14).

Last (Minor) Update: **June 10, 2013**. (2013.06.10)

You can write to us using this: online reply form. (It opens in a new window.)

MBR and Boot Records Index

PC **Assembly Code Index**

The Starman's Realm Index Page