DECODING Z80 OPCODES

- of use to disassembler and emulator writers -

Revision 2

Document by Cristian Dinu, compiled using various sources of information (see the acknowledgements section). You need a browser that supports Cascading Style Sheets (CSS) to view this file.

For any comments, suggestions, typo/factual error reports, please visit the World Of Spectrum forums and post a message there for GOC.

CONTENTS

---

1. INTRODUCTION

Instruction format and emulation notes

Z80 instructions are represented in memory as byte sequences of the form (items in brackets are optional):

[*prefix byte*,] *opcode* [,*displacement byte*] [,*immediate data*]
- OR -
two prefix bytes, *displacement byte*, *opcode*

The *opcode* (operation code) is a single byte whose bit pattern indicates the operation we need the Z80 to perform (register loading, arithmetic, I/O, etc.). The opcode may also contain information regarding the operation's parameters (operands), e.g. the registers which will be used/affected by the operation.

An optional *prefix byte* may appear before the opcode, changing its meaning and causing the Z80 to look up the opcode in a different bank of instructions. The prefix byte, if present, may have the values **CB**, **DD**, **ED**, or **FD** (these are hexadecimal values). Although there are opcodes which have these values too, there is no ambiguity: the first byte in the instruction, if it has one of these values, is always a prefix byte.

The *displacement byte* is a signed 8-bit integer (-128..+127) used in some instructions to specifiy a displacement added to a given memory address. Its presence or absence depends on the instruction

at hand, therefore, after reading the prefix and opcode, one has enough information to figure out whether to expect a displacement byte or not.

Similarly, *immediate data* consists of zero, one, or two bytes of additional information specifying explicit parameters for certain instructions (memory addresses, arithmetic operands, etc.). Its presence and number of bytes are also completely determined by the instruction at hand.

Note: Signed data is stored in 2's complement form. 16-bit data is stored LSB first.

A special class of instructions is accesed by using a **DD** or **FD** prefix, and then a **CB** byte. In this situation, the **CB** byte is also interpreted as a prefix, a *mandatory* displacement byte follows, and, finally, the actual opcode occurs. This is the situation that is described by the second byte pattern shown above.

Not all (prefix, opcode) combinations map to valid instructions. However, it is important to note that, unlike some other processors, upon encountering an invalid instruction, the Z80 will not 'crash' or signal an error - it will simply appear to do nothing (as if executing a **NOP** instruction), and continue with the next byte sequence in memory. There may also be several subtle effects, such as the temporary setting of some internal flags or the prevention of interrupts immediately after the read instruction. Invalid instructions are sometimes used to mark special commands and signals for emulators (e.g. Gerton Lunter's 'Z80' ZX Spectrum emulator).

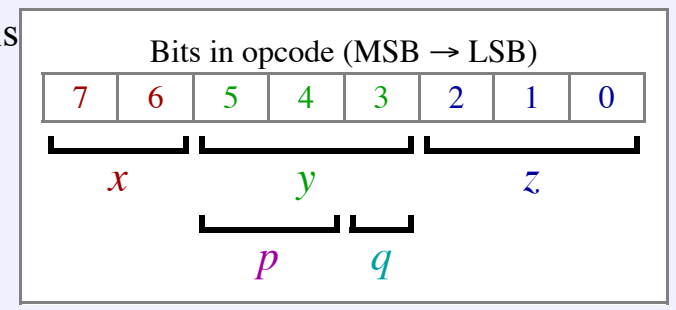
There may be several combinations of bytes that map to the same instruction. The sequences will usually have different execution times and memory footprints. Additionally, there are many instructions (not necessarily 'invalid') which do virtually nothing meaningful, such as **LD A, A**, etc., and therefore are reasonable substitutes for **NOP**.

Some instructions and effects are *undocumented* in that they usually do not appear in 'official' Z80 references. However, by now, these have all been researched and described in unofficial documents, and they are also used by several programs, so emulator authors should strive to implement these too, with maximal accuracy.

Finally, it is important to note that the disassembly approach described in this document is a rather 'algorithmic one', focused on understanding the functional structure of the instruction matrix, and on how the Z80 figures out what to do upon reading the bytes. If space isn't a concern, it is faster and easier to use complete disassembly tables that cover all possible (prefix, opcode) combinations - with text strings for the instruction display, and microcode sequences for the actual execution.

Notations used in this document

Upon establishing the opcode, the Z80's path of action is generally dictated by these values:



x = the opcode's 1st octal digit (i.e. bits 7-6)
y = the opcode's 2nd octal digit (i.e. bits 5-3)
z = the opcode's 3rd octal digit (i.e. bits 2-0)
p = *y* rightshifted one position (i.e. bits 5-4)
q = *y* modulo 2 (i.e. bit 3)

The following placeholders for instructions and operands are used:

d = displacement byte (8-bit signed integer)
n = 8-bit immediate operand (unsigned integer)
nn = 16-bit immediate operand (unsigned integer)

tab[x] = whatever is contained in the table named *tab* at index *x* (analogous for *y* and *z* and other table names)

Operand data may be interpreted as the programmer desires (either signed or unsigned), but, in disassembly displays, is generally displayed in unsigned integer format.

All instructions with *d*, *n* or *nn* in their expression are generally immediately followed by the displacement/operand (a byte or a word, respectively).

Although relative jump instructions are traditionally shown with a 16-bit address for an operand, here they will take the form **JR/DJNZ** *d*, where *d* is the signed 8-bit displacement that follows (as this is how they are actually stored). The jump's final address is obtained by adding the displacement to the instruction's address plus 2.

In this document, the 'jump to the address contained in HL' instruction is written in its correct form **JP HL**, as opposed to the traditional **JP (HL)**.

IN (C)/OUT (C) instructions are displayed using the traditional form, although they actually use the full 16-bit port address contained in BC.

In the expression of an instruction, everything in **bold** should be taken ad literam, everything in *italics* should be evaluated.

This document makes use of an imaginary instruction with the mnemonic **NONI** (No Operation No Interrupts). Its interpretation is 'perform a no-operation (wait 4 T-states) and do not allow interrupts to occur immediately after this instruction'. The Z80 may actually do more than just a simple NOP, but the effects are irrelevant assuming normal operation of the processor.

Disassembly tables

These tables enable us to represent blocks of similar instructions in a compact form, taking advantage of the many obvious patterns in the Z80's instruction matrix.

Table "r"

8-bit registers

Index	0	1	2	3	4	5	6	7
Value	B	C	D	E	H	L	(HL)	A

Table "rp"

Register pairs featuring SP

Index	0	1	2	3
Value	BC	DE	HL	SP

Table "rp2"

Register pairs featuring AF

Index	0	1	2	3
Value	BC	DE	HL	AF

Table "cc"

Conditions

Index	0	1	2	3	4	5	6	7
Value	NZ	Z	NC	C	PO	PE	P	M

Table "alu"

Arithmetic/logic operations

Index	0	1	2	3	4	5	6	7
Value	ADD A,	ADC A,	SUB	SBC A,	AND	XOR	OR	CP

Table "rot"								
Rotation/shift operations								
Index	0	1	2	3	4	5	6	7
Value	RLC	RRC	RL	RR	SLA	SRA	SLL	SRL

Table "im"								
Interrupt modes								
Index	0	1	2	3	4	5	6	7
Value	0	0/1	1	2	0	0/1	1	2

Table "bli"				
Block instructions				
Index[a,b]	**b=0**	**b=1**	**b=2**	**b=3**
a=4	LDI	CPI	INI	OUTI
a=5	LDD	CPD	IND	OUTD
a=6	LDIR	CPIR	INIR	OTIR
a=7	LDDR	CPDR	INDR	OTDR

---

2. UNPREFIXED OPCODES

FOR x=0						
z=0	**y=0**	**NOP**	**y=2**	**DJNZ** *d*		Relative jumps and assorted ops
	y=1	**EX AF, AF'**	**y=3**	**JR** *d*		
			y=4..7	**JR** *cc[y-4]*, *d*		
z=1	**q=0**	**LD** *rp[p]*, *nn*				16-bit load immediate/add
	q=1	**ADD HL,** *rp[p]*				
z=2	**q=0**	**p=0**	**LD (BC), A**	**p=2**	**LD** (*nn*), **HL**	Indirect loading
		p=1	**LD (DE), A**	**p=3**	**LD** (*nn*), **A**	
	q=1	**p=0**	**LD A, (BC)**	**p=2**	**LD HL,** (*nn*)	
		p=1	**LD A, (DE)**	**p=3**	**LD A,** (*nn*)	
z=3	**q=0**	**INC** *rp[p]*				16-bit INC/DEC
	q=1	**DEC** *rp[p]*				
z=4	**INC** *r[y]*					8-bit INC
z=5	**DEC** *r[y]*					8-bit DEC
z=6	**LD** *r[y]*, *n*					8-bit load immediate
z=7	**y=0**	**RLCA**	**y=4**	**DAA**		Assorted operations on accumulator/flags
	y=1	**RRCA**	**y=5**	**CPL**		
	y=2	**RLA**	**y=6**	**SCF**		

		y=3	**RRA**	y=7	**CCF**			
FOR x=1								
			LD *r[y]*, *r[z]*					8-bit loading
z=6		y=6	**HALT**					**Exception** (replaces **LD (HL), (HL)**)
FOR x=2								
			alu[y] r[z]					Operate on accumulator and register/memory location
FOR x=3								
z=0			**RET** *cc[y]*					Conditional return
z=1	**q=0**		**POP** *rp2[p]*					POP & various ops
	q=1	**p=0**	**RET**	**p=2**	**JP HL**			
		p=1	**EXX**	**p=3**	**LD SP, HL**			
z=2			**JP** *cc[y]*, *nn*					Conditional jump
z=3		y=0	**JP** *nn*	y=4	**EX (SP), HL**			Assorted operations
		y=1	*(CB prefix)*	y=5	**EX DE, HL**			
		y=2	**OUT** (*n*), **A**	y=6	**DI**			
		y=3	**IN A,** (*n*)	y=7	**EI**			
z=4			**CALL** *cc[y]*, *nn*					Conditional call
z=5	**q=0**		**PUSH** *rp2[p]*					PUSH & various ops
	q=1	**p=0**	**CALL** *nn*	**p=2**	*(ED prefix)*			
		p=1	*(DD prefix)*	**p=3**	*(FD prefix)*			
z=6			*alu[y] n*					Operate on accumulator and immediate operand
z=7			**RST** *y*8*					Restart

3. CB-PREFIXED OPCODES

x=0	*rot[y] r[z]*	Roll/shift register or memory location
x=1	**BIT** *y*, *r[z]*	Test bit
x=2	**RES** *y*, *r[z]*	Reset bit
x=3	**SET** *y*, *r[z]*	Set bit

4. ED-PREFIXED OPCODES

FOR x=0 OR x=3		
		Invalid instruction, equivalent to **NONI** followed by **NOP**
FOR x=1		
z=0	**y≠6** **IN** *r[y]*, **(C)** **y=6** **IN (C)**	Input from port with 16-bit address
z=1	**y≠6** **OUT (C)**, *r[y]* **y=6** **OUT (C), 0**	Output to port with 16-bit address
z=2	**q=0** **SBC HL**, *rp[p]* **q=1** **ADC HL**, *rp[p]*	16-bit add/subtract with carry
z=3	**q=0** **LD** (*nn*), *rp[p]* **q=1** **LD** *rp[p]*, (*nn*)	Retrieve/store register pair from/to immediate address
z=4	**NEG**	Negate accumulator
z=5	**y≠1** **RETN** **y=1** **RETI**	Return from interrupt
z=6	**IM** *im[y]*	Set interrupt mode
z=7	**y=0** **LD I, A** **y=1** **LD R, A** **y=2** **LD A, I** **y=3** **LD A, R** **y=4** **RRD** **y=5** **RLD** **y=6** **NOP** **y=7** **NOP**	Assorted ops
FOR x=2		
z≤3	**y≥4** *bli[y,z]*	Block instruction
		Otherwise, invalid instruction, equivalent to **NONI** followed by **NOP**

5. DD-PREFIXED OPCODES

If the next byte is a **DD**, **ED** or **FD** prefix, the current **DD** prefix is ignored (it's equivalent to a **NONI**) and processing continues with the next byte.

If the next byte is a **CB** prefix, the instruction will be decoded as stated in section 7, DDCB-prefixed opcodes.

Otherwise:

If the next opcode makes use of **HL**, **H**, **L**, *but not* ***(HL)***, any occurrence of these will be replaced by **IX**, **IXH**, **IXL** respectively. An exception of this is **EX DE, HL** which is unaffected.

If the next opcode makes use of **(HL)**, it will be replaced by **(IX+*d*)**, where *d* is a signed 8-bit displacement immediately following the opcode (any immediate data, i.e. *n*, will follow the displacement byte), and any other instances of **H** and **L** will be unaffected. Therefore, an instruction like **LD IXH, (IX+*d*)** does not exist, but **LD H, (IX+*d*)** does.

All other instructions are unaffected.

---

6. FD-PREFIXED OPCODES

The **FD** prefix acts exactly like the **DD** prefix, but the **IY** register is used instead of **IX**.

Note that there is no way to "mix" prefixes so that we access both **IX** and **IY** within the same operation: an instruction like **LD IXH, IYH** does not exist. As stated previously, if two **DD/FD** prefixes appear in succession, e.g. **FD DD** *opcode*, only the last one will be taken into consideration.

---

7. DDCB/FDCB-PREFIXED OPCODES

These instructions have the following format:

DD *or* **FD** *prefix*, **CB**, *displacement byte*, *opcode*

CB and *opcode* will form instructions similar to those in the CB-prefixed opcodes section above. However, these will now operate on **(IX+*d*)** and, if the instruction isn't **BIT**, copy the result to the register they would have initially acted upon, unless it was **(HL)**.

In the case of the **DD** prefix, the instruction table is thus:

x=0	**z≠6** **LD** *r[z]*, *rot[y]* **(IX+*d*)** **z=6** *rot[y]* **(IX+*d*)**	Roll/shift memory location and copy result to register
x=1	**BIT** *y*, *r[z]*	Test bit at memory location
x=2	**z≠6** **LD** *r[z]*, **RES** *y*, **(IX+*d*)** **z=6** **RES** *y*, **(IX+*d*)**	Reset bit and copy result to register
x=3	**z≠6** **LD** *r[z]*, **SET** *y*, **(IX+*d*)** **z=6** **SET** *y*, **(IX+*d*)**	Set bit and copy result to register

An instruction such as **LD** *r*, **RES** *b*, **(IX+*d*)** should be interpreted as "attempt to reset bit *b* of the byte at **(IX+*d*)**, and copy the result to register *r*, *even the new byte cannot be written at the said address* (e.g. when it points to a ROM location).

Such an instruction is sometimes also represented in this form: **RES** *r*, **(IX+*d*)**, *r*.

---

8. ACKNOWLEDGEMENTS

The 'algorithm' described herein was constructed by studying an "instruction/flags affected/binary form/effect" list in a Romanian book called "Ghidul Programatorului ZX Spectrum" ("The ZX Spectrum Programmer's Guide").

The exact effects and quirks of the CB/DD/ED/FD prefixes, as well as the undocumented ED and CB instructions, were learnt from "The Undocumented Z80 Documented" by Sean Young.

My sincere thanks to all those who have contributed with suggestions or corrections. They are mentioned in the following section.

---

9. REVISION HISTORY

Revision 1

Implemented a better representation for the **DDCB** instructions (thanks to Ven Reddy) and for certain "invalid" **ED** instructions, e.g. a more accurate **NOP/NONI** instead of an 8T **NOP** (thanks to Dr. Phillip Kendall). Fixed some typos.

Revision 2

Radically altered the presentation. Added an intro section, some diagrams, more comments and a Revision History section. Fixed some important typos and changed the wording in some places to avoid misunderstandings (thanks to BlueChip for his numerous and helpful suggestions; he also suggested that I add info on the signed number format and byte order).

- EOF -