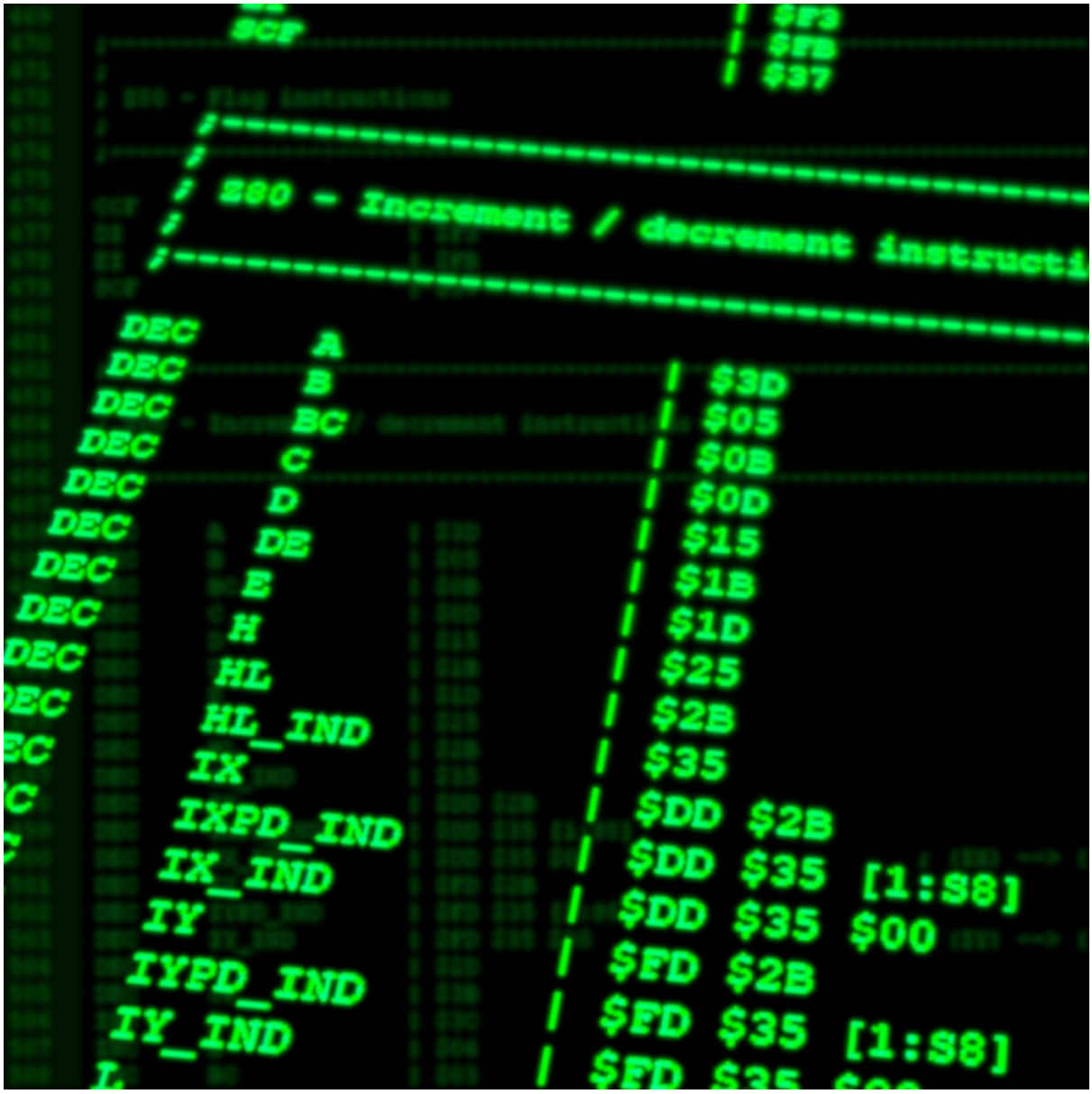
XA80 Opcode Compiler
User Manual

V0.3



Contents

Document Release Status

Version	Date	Changes
0.3 R1	08-Jul-2023	Initial release of V0.3 of the manual

Contact

For contact about the content of this document, please contact Duncan Munro:

duncan@duncanamps.com

The software, and this document, can be obtained from:

https://github.com/duncanamps/xa80

1. Introduction

1.1. Disclaimer

XA80 is experimental open source software and is not guaranteed to work correctly in all conditions.

All trademarks are acknowledged as belonging to their respective owners.

1.2. Document purpose

This document is the User Manual to cover the Opcode Compiler for XA80, **X** (Cross) **A**ssembler for x**80** processors. Its purpose is to provide a reference on how the application should be used, with examples where appropriate.

1.3. Application scope

XA80 is a multi-platform cross assembler available for Windows, Linux and MacOS. It is intended to be used with the following 8/16 bit processors:

- 8080
- 8085
- Z80
- Z180

The Opcode Compiler is companion tool which generates tables for the cross assembler to use. The processors listed above are “baked in” to the assembler, however it is possible to create additional sets of opcodes over and over these for a number of reasons:

- The ability to accommodate a related processor with a slightly different instruction set
- The ability to enable undocumented features and instructions of one of the processors listed above

1.4. Related documents

XA80 Cross Compiler User Manual

2. Opcode Compiler operation

The Opcode Compiler takes a textual list of opcodes normally having a **.opcode** filetype. This is then compiled into a compact binary form that can be used by the assembler, and these will have a **.opcode.bin** filetype.

2.1. Command line usage

The command line usage of the Opcode Compiler is as follows:

```
oc_comp myfile.opcode
```

This will create the output file **myfile.opcode.bin**

It's possible to use path structures and the compiler will respect this:

```
oc_comp source\z80.opcode
```

The above will create the output file **source\z80.opcode.bin**

2.2. Other command line options

Some other command line options which can be used are:

```
oc_comp --help
oc_comp myfile.opcode --verbose
```

Using the command line switch **--help** will provide limited help on the operation of the software.

The command line switch **--verbose** will provide a detailed listing of the mnemonics and instructions that have been compiled from the source .opcode file.

2.3. Using with the assembler

To use the newly created opcode.bin file with the assembler, move it alongside the executable code for the assembler – this is the only place the assembler will look for it.

For example, if you create a file **upd780c.opcode.bin**, please move it to the folder containing **xa80.exe** or **xa80** depending on your operating system. The new opcode file can be accessed by the following assembler command line:

```
xa80 myfile.asm –com –processor=upd780c
```

3. .opcode file format

The .opcode file is a text file and will typically contain:

- Blank lines
- Comments
- Definitions

3.1. Comments

Comments can appear anywhere in the file and are indicated by a `;` semicolon. Anything after the semicolon is ignored so the comments can appear after a definition:

```
;
; Comment
; blank line follows

DEC     IX_IND        | $DD $35 $00                       ; (IX) --> (IX+0)
```

3.2. Definitions

Definitions are split into two field groups, the first is the opcode and operands and the second is the code generation part. The two field groups are separated by a `|` vertical brace character.

3.3. Opcodes and operands

The opcode is the mnemonic that will be used by the assembler to access a specific instruction. These are not case sensitive.

Examples are **`LDIR`**, **`CP`**, **`XOR`**.

The operands are what comes after the opcode, and can be numbers or register names or some combination of the two. In the .opcode file definition, there are no commas between the operands, just spaces or tabs. The following table lists what can be used as an operand:

.OPCODE	.ASM FILE
A	A
AF	AF
AF_	AF'
B	B
BC	BC
BC_IND	(BC)
C	C
C_IND	(C)
D	D
DE	DE
DE_IND	(DE)
E	E
F	F
H	H
HL	HL
HL_IND	(HL)
I	I
IX	IX
IXH	IXH
IXL	IXL
IX_IND	(IX)
IXPD_IND	(IX+NN)

.OPCODE	.ASM FILE
IY	IY
IYH	IYH
IYL	IYL
IY_IND	(IY)
IYPD_IND	(IY+NN)
L	L
M	M
NC	NC
NZ	NZ
P	P
PE	PE
PO	PO
PSW	PSW
R	R
SP	SP
SP_IND	(SP)
U8	NN
U8_IND	(NN)
U16	NNNN
U16_IND	(NNNN)
Z	Z

In the above, the left hand column is how an operand will be shown in a **`.opcode`** file, while the right hand column shows typically how it would be presented in an assembler source file.

Most of the register names, such as **`B`**, **`C`**, **`HL`** will be familiar. Other items such as the **`_IND`** suffix indicate an indirection. So **`HL`** indicates the contents of the HL register while **`HL_IND`** -> **`(HL)`** indicates the contents of the address pointed to by HL.

`U8` is an unsigned 8 bit value, while **`U16`** is an unsigned 16 bit value. **`IXPD`** and **`IYPD`** indicate IX Plus Displacement and IY Plus Displacement respectively.

Please note that IXH, IXL, IYH, IYL, IX_IND and IY_IND are not normally Z80/Z180 register operands. They are used to support undocumented instructions and alternative instructions.

The table shown in this section is fixed; that is to say, extra operands cannot be added without recompiling the opcode compiler, the assembler, and making alterations to the file format.

3.4. Code generation

After the `|` vertical brace comes the code generation part. This consists of one or more definitions, each one of which will produce a byte of output code.

The full table of code generation definitions is as follows:

CODE GEN DEF	OUTPUT
$xx	A hex value, e.g. $ED or $57
xxx	A decimal value, e.g. 123
%xxxxxxxx	A binary value, e.g. $10100110
[p:R8]	An 8 bit value as a relative ranged from ORG+2 where p is 1 or 2 to represent operand 1 or operand2
[p:U8]	An 8 bit unsigned value
[p:S8]	An 8 bit signed value, used with (IX+nn) etc.
[p:U16]	A 16 bit unsigned value
[p:B3]	A 3 bit value which can be output on its own or embedded in a binary value. For example, %101[1:B3]10. If operand 1 is binary 011 the generated output will be %10101110.
[p:RST]	RST number in 3 bits where 000 -> RST $00, 001 -> RST $08, 010 -> RST $10 etc.
[p:IM]	IM 0/1/2 where operand p generates 0->$46, 1->$56, 2->$5E

3.5. Examples

Here are some real examples, taken from the **`z80.opcode`** file:

```
SCF                 | $37
```

This is an easy example; it has no operands and just a mnemonic with one code generation definition.

On encountering the **`SCF`** mnemonic with no operands, the assembler will generate a hex output of **`$37`** to signify the Set Carry Flag instruction.

```
JP      U16         | $C3 [1:U16]
```

This example shows the mnemonic with one operand, in this case JumP to an unsigned 16 bit address. The assembler on encountering the JP mnemonic with one numeric expression will output hex $C3 followed by an unsigned 16 bit value for the address. In this case three bytes will be output, the $C3 followed by two bytes in little-endian format for the 16 bit address.

An assembler line of **`JP $1234`** will output **`$C3 $34 $12`**.

```
DEC     IXPD_IND    | $DD $35 [1:S8]
```

Another single operand, this time using a signed 8 bit value. The expression used with (IX+… is evaluated and must be in the range -128 to +127.

```
LD      IYPD_IND U8   | $FD $36 [1:S8] [2:U8]
```

Finally, an example with two operands. The first is the (IY+… and the second is an unsigned 8 bit literal value. Consider the following assembler code:

```
offset      EQU   6
newval      EQU   200
loop:       LD    (IY+offset),newval
```

In this assembler snippet, the instruction would generate an output of **`$FD $36 $06 $C8`**.

3.6. Alternatives

It is possible to put in more than one definition for the same instruction. This allows alternative grammar to be used within the assembler. For example the following two definitions from an opcode file allow different assembler input but will both generate the same code:

```
AND         A C   | $A1
AND         C     | $A1               ; Alternative version of AND A,C
```

Another alternative is with the IX/IY plus displacement instructions. These indirect instructions much always have a displacement coded, even if it's zero. The following shows how it's possible to use LD (IY),0 even though it is not a Z80 instruction:

```
LD          IYPD_IND U8   | $FD $36 [1:S8] [2:U8]
LD          IY_IND U8     | $FD $36 $00 [2:U8]  ; (IY) --> (IY+0)
```

Notice how the second line allows (IY) without the displacement, but codes in a $00 byte in third place automatically.

3.7. Undocumented instructions

The file z80x.opcode contains the normal Z80 instructions with the addition of hundreds of lines for undocumented instructions. Some examples are:

```
CP      IXH       | $DD $BC            ; *UNDOCUMENTED*
SET1LDH IXPD_IND  | $DD $CB [1:S8] $CC ; *UNDOCUMENTED* SET 1,(IX+N)
                                       ; then LD H,(IX+N)
```

The first line uses an existing mnemonic with an operand of IXH which is not normally used in Z80 assembler code. It should be noted that only the operands listed in section 3.3 can be used.

The second line uses a new mnemonic of SET1LDH – this is completely made up and represents “set bit 1 of (IX+displacement) then load the contents of (IX+displacement) to the H register”.

You can freely devise your own mnemonics to support undocumented or new instructions, however please bear in mind that this is likely to render your assembler files unusable by others.

4. Appendices

4.1. Appendix – Opcodes

The following opcodes are defined by the application, however you can add to these if required:

Opcode	8080	8085	Z80	Z180
ACI	Y	Y		
ADC	Y	Y	Y	Y
ADD	Y	Y	Y	Y
ADI	Y	Y		
ANA	Y	Y		
AND			Y	Y
ANI	Y	Y		
BIT			Y	Y
CALL	Y	Y	Y	Y
CC	Y	Y		
CCF			Y	Y
CM	Y	Y		
CMA	Y	Y		
CMC	Y	Y		
CMP	Y	Y		
CNC	Y	Y		
CNZ	Y	Y		
CP	Y	Y	Y	Y
CPD			Y	Y
CPDR			Y	Y
CPE	Y	Y		
CPI	Y	Y	Y	Y
CPIR			Y	Y
CPL			Y	Y
CPO	Y	Y		
CZ	Y	Y		
DAA	Y	Y	Y	Y
DAD	Y	Y		
DCR	Y	Y		
DCX	Y	Y		
DEC			Y	Y
DI	Y	Y	Y	Y
DJNZ			Y	Y
EI	Y	Y	Y	Y
EX			Y	Y
EXX			Y	Y
HALT			Y	Y
HLT	Y	Y		
IM			Y	Y

Opcode	8080	8085	Z80	Z180
IN	Y	Y	Y	Y
IN0				Y
INC			Y	Y
IND			Y	Y
INDR			Y	Y
INI			Y	Y
INIR			Y	Y
INR	Y	Y		
INX	Y	Y		
JC	Y	Y		
JM	Y	Y		
JMP	Y	Y		
JNC	Y	Y		
JNZ	Y	Y		
JP	Y	Y	Y	Y
JPE	Y	Y		
JPO	Y	Y		
JR			Y	Y
JZ	Y	Y		
LD			Y	Y
LDA	Y	Y		
LDAX	Y	Y		
LDD			Y	Y
LDDR			Y	Y
LDI			Y	Y
LDIR			Y	Y
LHLD	Y	Y		
LXI	Y	Y		
MOV	Y	Y		
MLT				Y
MVI	Y	Y		
NEG			Y	Y
NOP	Y	Y	Y	Y
OR			Y	Y
ORA	Y	Y		
ORI	Y	Y		
OTD				Y
OTDM				Y
OTDMR				Y

Opcode	8080	8085	Z80	Z180
OTDR			Y	Y
OTI				Y
OTIM				Y
OTIMR				Y
OTIR			Y	Y
OUT	Y	Y	Y	Y
OUT0				Y
OUTD			Y	Y
OUTI			Y	Y
PCHL	Y	Y		
POP	Y	Y	Y	Y
PUSH	Y	Y	Y	Y
RAL	Y	Y		
RAR	Y	Y		
RC	Y	Y		
RES			Y	Y
RET	Y	Y	Y	Y
RETI			Y	Y
RETN			Y	Y
RIM		Y		
RL			Y	Y
RLA			Y	Y
RLC	Y	Y	Y	Y
RLCA			Y	Y
RLD			Y	Y
RM	Y	Y		
RNC	Y	Y		
RNZ	Y	Y		
RP	Y	Y		
RPE	Y	Y		
RPO	Y	Y		

Opcode	8080	8085	Z80	Z180
RR			Y	Y
RRA			Y	Y
RRC	Y	Y	Y	Y
RRCA			Y	Y
RRD			Y	Y
RST	Y	Y	Y	Y
RZ	Y	Y		
SBB	Y	Y		
SBC			Y	Y
SBI	Y	Y		
SCF			Y	Y
SET			Y	Y
SHLD	Y	Y		
SIM		Y		
SLA			Y	Y
SLP				Y
SPHL	Y	Y		
SRA			Y	Y
SRL			Y	Y
STA	Y	Y		
STAX	Y	Y		
STC	Y	Y		
SUB	Y	Y	Y	Y
SUI	Y	Y		
TST				Y
XCHG	Y	Y		
XOR			Y	Y
XRA	Y	Y		
XRI	Y	Y		
XTHL	Y	Y		

4.2. Appendix – Binary format of .opcode.bin files

This section describes the format of the .opcode.bin files. The general format is a header followed by 1 or more records listing the mnemonics and finally 1 or more records describing each instruction entry.

4.2.1..opcode.bin header

The header consists of the following entries:

ENTRY	BYTES	NOTES
Magic word	4	Contains $4D43504F which is OPCM backwards (OPCode Map)
File_version	2	The file version (currently 2)
Mnemonic_recs	2	Number of mnemonic records
Instruction_recs	2	Number of instruction records
Checksum1	2	The above 5 word items XORed together

All integer values are little-endian, so if the number of records was 260, it would be stored as hex $04 $01.

4.2.2..opcode.bin mnemonics

The next section is a table of the mnemonics. The header described in section 4.2.1 provides the number of records that will be in this section. Each record has the following format:

ENTRY	BYTES	NOTES
Mnemonic	9	Up to 8 ASCII characters for the mnemonic. All remaining bytes are padded with 0 bytes

4.2.3..opcode.bin instructions

The final section is the table of instructions:

ENTRY	BYTES	NOTES
Mnemonic_index	2	16 bit value in the range 0..n-1 where n is the index into the mnemonic table described in section 4.2.2
Operand1_index	1	8 bit value providing the index of the operand no. 1. For example, 0=Undefined, 1=A, 2=AF, 3=AF', 4=B, etc. The full list is given in section 3.3. If the operand is not given, the value will be 0 (undefined)

Operand2_index		1	Same as operand 1 but for operand 2
Code_elements		1	Count of code elements, typically 1 to 4
Code_elem0	Type	1	0=Null 1=B3 2=Hex 3=IM 4=R8 5=S8 6=RST 7=U8 8=U16
	Operand	1	Either 1 or 2, 0 if not used / relevant
	Value	1	The overall value of the byte code. This will be the hex byte, or bits to which other items may be encoded into
	Offset	1	The offset of a 3 bit value within a byte. A value of 0 indicates that that the three bit value occupies bits 2,1,0. A value of 5 indicates that the three bit value occupies bits 7,6,5
Code_elem1	If used, same definition as Code_elem0		
More…	If used		