Z80 Generation

Paul Robson September 2021

This document describes the quasi macro language used to generate the Z80 assembler sources.

Instructions are generated from lines with the following format

```
[override]    <Base Opcode>     “<Mnemonic>”
    <Code>
    <Code>
```

This creates one or more opcodes that are stored in the table of opcodes and mnemonics.

The table consists of the following groups

1. The default group (e.g. the 8080 set)
2. The CB group (bits, shifts, rotates)
3. The ED group (extended instructions)
4. The DD group (IX instructions)
5. The DDCB group (bits shifts and rotates on IX)

IY is implemented by IX. In fact, we use an imaginary pointer “IZ” which is written in code where one would write IX or IY, which is a pointer to either IX or IY, which are functionally equivalent.

The *override* keywords stops the normal behaviour which is to throw out duplicate code. This can be useful for things like writing back to HL. One may want to generate (say) B = FETCH() for LD B,nn but WRITE16(HL(),FETCH()) for LD (HL),nn even though they are the same group.

Each opcode can contain one or more “substituter”. These are words prefixed with @, so in the above example it may be @tgtreg (for target register). These count sequentially and provide different replacements for each replacer, all of which default to the replacer.

Examples are clearer. In the example LD ABCDEHL,nn which is one instruction with varying targets

```
    @tgtreg [ shift:3,B,C,D,E,H,L,_,A ]
```

We have defined a replacement which ignores (HL) which is a special case, so this will create 7 instructions , with the offset (0,1,2,3,4,5 and 7) shifted left 3 times. We generate this instruction with the following instructions

```
    06   “LD @tgtreg,$1”
         @tgtreg = FETCH8()

    36 “LD (HL),$1”
         WRITE8(HL(),FETCH())
```

Some instructions do not need the variation. One could define

```
    @srcreg [ Shift:0, B,C,D,E,H,L,(HL):READ8(HL()),A ]
```

for things like ADD A,r Note that the HL now has *two* replacers, the first (HL) is used in the mnemonic and the second READ8(HL()) is used in the code.

One can use more than one LD r,r’ for example could use both of these, leaving just LD (HL),r to be generated seperately.

```
    40   “LD @tgtreg,@srcreg”
         @tgtreg = @srcreg

    70   “LD (HL),@srcreg”
         WRITE8(HL(),@srcreg)
```

Instruction groups can be produced wholesale using macros e.g.

```
    @aluop [ Shift:3 ADD,ADC,SUB,SBC,AND,OR,XOR,CMP ]

    80 “@aluop,@srcreg”
         @aluop(@srcreg)
```

Some parts of things like the BIT n,r SET n,r and RES n,r could be done by three, one for the instruction, one for the bit number, and one for the operating register, though SET and RES on (HL) would be special cases because of the write back.

Instructions are generated from a ruby script which takes a list of such source files and produces a 256 entry mnemonic and code table pair for each instruction group (in the order specified earlier).

Definitions are held in a subdirectory “defines” and are read in at the start.

Comments in all files are where the line starts with #

Timings are not supported, but each instruction will be given the average cycles, with a specific add on for the ‘instruction shifts’.