**MZ Language**

MZ is a Z80 based FORTH derivative. The main problem with FORTH on a Z80 is the twin stacks ; a Z80 only has one stack, the return stack, so much of the work has to be done by one of the other registers which move data in 8 bits.

MZ solves this problem by replacing a stack with three 16 bit registers, known as A, B and C. These registers directly map (currently) to HL, DE and BC.

The MZ language consists of words like FORTH. Unlike FORTH, every word is treated identically, though they do not of course behave identically.

MZ code is simply a sequence of MZ words, or decimal constants, or string constants

The decimal and string constants are put on the ‘stack’ using the sequence ex de,hl ; ld hl,<constant>. For strings, the value put is a pointer to an ASCIIZ string.

**Words**

Words are straight Z80 code, ending in a RET (or sometimes a JP rather than a CALL/RET etc.). Their functionality is defined by the lower 4 bits in the type byte in the dictionary.

**Standard Words (type 0)**

Standard words are normal Z80 code. To execute a word, you call it. To compile a word, you compile a call to it ; this is either a CD xx xx Z80 call, or a call that involves a page switch.

**Immediate words (type 15)**

Immediate words are like FORTH, they execute whether they are compiled or interpreted. They are used for words that are run on the fly, such as “if” “begin” “private” and “variable”.

**Variable words (type 14)**

Variable words convert the address to a constant (see constants), so the effect of the code is to put the current address into A (and do A→B first). Variables are private by default as they can only be accessed by routines in their own page of memory. This requires a style of code where variables are local to their module and accessor/mutator functions are used to access them from elsewhere.

**Code Copying words (type 1-10)**

Code copying words are an optimisation. For speed reasons, and sometimes for space reasons, and sometimes for both, code can be ‘inlined’. A code copying word consists of a number of bytes, which is the same length as the type number followed by a ret. The ret is so the code copying word can actually just be normally executed as for standard words, (subject to the protected flag)

e.g. + would be type 1 and would look like this

add hl,de ; this one byte is the actual code.

ret ; this ret is present so it can be executed

The user cannot create these (or to be precise, isn’t allowed to). They can assumed to be in unpaged memory, so do not have to be read using far memory access.

**Constants**

Besides words, the language contains constants, which are decimal integers, possibly prefixed with a ‘-’. These produce code that load a constant.

When you load a constant into the register set it always goes in A, but before it does that, it ***transfers the contents of A into B***. So constant 3 actually generates this code

ex de,hl ; transfer A->B

ld hl,3 ; load 3 into A.

The reason for this oddity is that it allows the fluency of FORTH, MZ’s parent language. You can write , as you can in FORTH.

count @ 2 \* 3 + count !

and it will work just the same (though MZ can do it better). Where it falls apart is when you need to use three values ; because then you have to use C or the return stack as a working space. (words like a>b b>c r>c exist for this purpose).

When I first conceived of this I thought it likely it would lead to an explosion of variables, but actually it doesn’t, though there are certainly more variables.

This technique sounds inherently dumb, but it actually works quite well at a slight cost in space and speed for readability. A previous version replaced the ex de,hl with a function which did the same thing when required, and the readability of the code went down significantly.

There are also string constants ; these begin with a “ mark. They put a constant (as above) which points to an ASCIIZ representation of the rest of the word, with any underscores replaced by spaces.

**Language Environment**

There are two operating modes ‘compile’ and ‘execute’ switch between by those two words. Nothing else changes these modes.

In compile mode, everything is compiled, except words marked as immediate which are executed.

In execute mode, everything is executed (the code to load constants is simulated). There are two limitations. Execute mode does not support string constants which can only exist in compiled code, and certain other words are marked *protected*, see below.

Dictionary words can be marked as *private*. This means that they are not added to the internal dictionary at the end of a module compilation.

Dictionary words can also be marked as *protected.* This means they cannot be executed by the user typing in to the console or as part of a text being loaded, because they’d probably crash it – these are words that manipulate the return stack for example.

When you are typing in code at the console rather than via the screen editor, or loading it in via bootstrap, it is done in execute mode automatically. You can switch to compile mode if you want, if so, be it on your own head.

**Dictionary**

|  |  |
| --- | --- |
| Offset | Contents |
| +0 | Offset to next entry, normally word length + 5, 0 at end |
| +1 | Page number of word |
| +2,+3 | Address of word |
| +4 | Type (0..3) Private (7) Protected (6) |
| +5 | First character of word name (ASCII) |
| +n | Last character of word name (ASCII) bit 7 is set |

**Core Words**

|  |  |  |  |
| --- | --- | --- | --- |
| Word | Com? | Pro? | Description |
| - |  |  | B – A → A |
| ; |  | Yes | Return from subroutine |
| : <name> | Yes |  | Define a word. |
| ! |  |  | 16 bit write B to memory locations A,A+1 |
| !! @@ && | Yes |  | Variable operators |
| .hex |  |  | Print word on console |
| [-]if .. then | Yes |  | Conditional execution |
| @ |  |  | 16 bit read of A → A |
| \* |  |  | A \* B → A |
| / |  |  | B / A → A |
| + |  |  | A + B → A |
| +! |  |  | Add B into memory at A,A+1 |
| < = > <> >= <= |  |  | Comparison B vs A → A = -1 if true, - if false |
| 0- |  |  | 2’s complement A |
| 0< |  |  | A is -1 if A <0, 0 otherwise |
| 0= |  |  | A is -1 if A == 0, 0 otherwise |
| 1+ 1- 2+ 2- |  |  | Increment/Decrement A |
| 2\* 4\* 8\* 16\* 256\* |  |  | Shift A left |
| 2/ 4/ 16/ 256/ |  |  | Arithmetic Shift A right |
| a>b b>a b>c etc |  |  | Register→Register transfer |
| a>r abc> b>r c>r |  | Yes | Push one or all registers on stack |
| abs |  |  | |A| → A |
| and |  |  | A & B → A |
| begin .. [-]until | Yes |  | Conditional loop |
| begin .. again | Yes |  | Unconditional loop |
| break |  | Yes | Compile a CSpect break |
| bswap |  |  | Swap upper/lower bytes of A |
| c! |  |  | Save B at memory location A (8 bits) |
| c@ |  |  | Load A from memory location A (8 bits) |
| copy |  |  | Move C bytes from B to A, copes with overlap. |
| debug |  |  | Display ABC on console |
| fill |  |  | Fill C bytes of memory with B starting at A |
| for .. next | Yes |  | Loop repeated a constant number of times. Note, 5 for next counts 4,3,2,1,0 and index in A at the start of every loop. |
| halt |  | Yes | Stop Z80 working |
| i | Yes |  | Get loop index into A, A → B |
| inkey |  |  | Read current pressed key or 0 if none |
| gfx.write.char |  |  | Print character E in colour D (Spectrum colours) at position HL |
| gfx.initialise |  |  | Initialise display mode, clear screen and border, home .hex cursor. Different kernels have different graphics support. |
| max |  |  | Larger of A,B → A |
| min |  |  | Smaller of A,B → A |
| mod |  |  | Remainder of B / A → A |
| module  endmodule | Yes |  | Brackets module code. All it does is throw an error if the module starts and ends in a different page of memory. |
| not |  |  | One’s complement A |
| or |  |  | A | B → A |
| p! |  |  | Write B to port A (16 bit address) |
| p@ |  |  | Read A from port A (16 bit address) |
| private | Yes |  | Mark last definition as private |
| protected | Yes |  | Mark last definition as protected |
| r>a r>abc r>b r>c |  | Yes | Pop one or all registers off the stack. |
| swap |  |  | Swap A and B |
| sys.info |  |  | System Information address → A, A → B |
| variable <name> | Yes |  | Make last definition a variable accessed via && !! and @@ |
| xor |  |  | A ^ B → A |