ccz80 language specification
version 2.0.8

Cross Compiler for Z80

http://www.telefonica.net/web2/emilioguerrero/ccz80/ccz80.html

Contents

Preliminary

This is a compiler for create assembler code and binary code from a source code in ccz80 language for a Zilog Z80 based computer (Amstrad CPC, Amstrad PCW, Spectrum 16/48/+2/+3, MSX system, ...).

The compiler is developed in C# language, so you need install Microsoft .NET Framework 2.0 or later in your computer for use ccz80 or use WINE or MONO utility in Linux or MONO in MacOS.

For use this compiler your need do the next steps in your system (PC, Mac, etc.):

1. Write a program in ccz80 language and save in a text file (usually with .ccz80 extension).

2. Compile the program in command line using the syntax:

```
ccz80 <source file> [/org=<address>] [/asm]
      [/include=<path-list>] [/post=<aplication>] …]
```

where <source file> is the source program write and saved in first moment, optionally <address> is the start address for the assembler object code created (this address is 0 if not specified). If /asm option is specified the compiler only generate the assembler code equivalent to source ccz80 code, and don't generate the binary file. With /include option the compiler search the files in include and datafile sentences in all paths in <path-list> (separate by semicolon). You can specify several /post options to run aplications, with parameters, after the compilation process, if it is correct; the applications are run in order as specified in command line and only an application is running if the previous one products a return value 0 (it's recommended enclose each option /post=<aplication> in inverted commas if use spaces to parameter separation).

3. Load and execute the binary file created with the same name as the source code and .bin extension in an emulator o real Z80 based computer. The compiler creates too the the assembler code in a file with the same name as the source code and .asm extension.

The ccz80 compiler returns a ERRORLEVEL value 0 if the compilation is correct, and 1 if it isn't.

ccz80 language description

Data types

- byte: number in range 0 to 255.
- word: number in range 0 to 65535.

You can use a value byte or word as a pointer using the * and ** operators.

The conversion between both types is automatic in any expression.

Float number type don't exist. It's necessaty create a set of functions for convert and operate with this data type.

String type don't exist. You need declare an array byte with lenght maximum characters to content plus one for the final character 0 added to all strings. In the standard library there are some functions for manipulate this type of string.

It's important note that an operation produces a negative value or you use the unary minus operator (-) the result is the positive value two complemented. By example:

```
byte b = 5;
word w;

b -= 9; // The expression produces -4, but the b value is 252 (256 - 4).
w = -8; // Produces -8, but the w value is 65528 (65536 - 8).
return;
```

Elements

- Single line comments: starting with //.
- Keywords: include, asm, byte, word, array, function, if, else, do, while, for, repeat, goto, gosub and return. The keywords are case sensitive, the others identifier are not (labels, variables and functions).

- Labels: identifier starting with a letter a to z or a digit 0 to 9, follow by a letter, digit or underscore character.

- Expressions.

Expression elements

- Brackets: (and).

- Unary operators: Same signification and use as C language.

  ++ (pre and post-operand)
  -- (pre and post-operand)
  !
  -
  *
  ** (get a word value from the address specified by the operand, same priority as *)
  &
  ~

- Binary operators: Same signification and use as C language.

  *
  /
  %
  +
  -
  <<
  >>
  <
  <=
  >
  >=
  ==
  !=
  &
  ^
  |
  &&
  ||
  =
  *=
  /=
  %=
  +=
  -=
  <<=
  >>=
  &=
  ^=
  |=

- Conditional operator.

  ? and : (evaluate ever true and false parts. Its recommended enclose in brackets (and) the condicional expressions)

(All operators with the same signification and priority in language C.)

- Byte variables.
- Word variables.
- Byte constant: in decimal or hexadecimal with the prefix #. By example 10 and #0A are the same value.
- Character constant: equavalent to a constant byte, it's a character enclose in single quotation marks. For specifier special characters it's possible use the escape sequences:

  \0 (for ASCII character 0)
  \a (for ASCII character 7)
  \b (for ASCII character 8)
  \e (for ASCII character 27)
  \f (for ASCII character 12)
  \n (for ASCII character 10)
  \r (for ASCII character 13)
  \t (for ASCII character 9)
  \v (for ASCII character 11)
  \' (for ASCII character 39)
  \" (for ASCII character 34)
  \\ (for ASCII character 92)
  \xnn (for ASCII character n, where n is specified in hexadecimal)

- Word constant: in decimal or hexadecimal with the prefix #. By example 4096 and #1000 are the same value.
- Constant string: a character list enclose in double quotation marks. It's possible use the same escape sequences that in a character constant.
- Label: defined as an array or as a point in the program code, take the value where the array is located in memory or the code address where the label is defined.
- Function: with the params, take the return value of the funcion.

Variable arrays don't exist. You can implement an array with pointers making:

```
array byte array_byte[10]; // Declares a 10 elements array of bytes
byte n;
word i = 5; // Declares a word variable for index of array
n = *(array_byte + i); // Assigns to n the sixth element of array
```

If the array is of word type, the implementation of same example is:

```
array word array_word[10]; // Declare a 10 elements array of words
word n;
word i = 5; // Declares a word variable for index of array_word
n = **(array_word + i * 2); // Assigns to n the sixth element of array
```

For an byte type two dimensions array:

```
array word array_byte[40]; // Declare 40 elements for an array
                              of 4 rows x 10 columns
byte n;
byte r = 2, c = 7; // Declares variables for indexes of row
                      and column of array
n = *(array_byte + r * 10 + c); // Assigns to n the eighth element of
                                   thirth row of array
```

And for an word type two dimensions array:

```
array word array_word[40]; // Declare 40 elements for an array
                              of 4 rows x 10 columns
word n;
byte r = 2, c = 7; // Declares variables for indexes of row
                      and column of array
n = **(array_word + (r * 10 + c) * 2); // Assigns to n the eighth
                                          element of thirth row
                                          of array
```

Control structures and sentences

include <string>, ... ;

Insert in the place where is specified the source file/s defined by the <string> list. A file is include in program only one time, if it's included another time, it's ignored.

datafile <string> = <identifier>|<word constant>, ... ;

Define at program end the label <identifier> and place next the content of the binary file <string>, or place the content in the address specified by <word constant>.

asm { <string>, ... }

Insert in the place where is specified the assembler instructions defined by the <string> list.

const <identifier> = <word constant>, ... ;

Define the constant <identifier> for use instead a byte constant or word constant in any place of rest of program (declarations, expression, etc.) and the others files that will be included with include sentence.

byte <identifier> [= <byte constant>|<character constant>], ... ;

Declare the byte variables specified by <identifier> (same rules in that labels definition), and optionally define initial values. If the initial value is not defined it's zero.

word <identifier> [= <word constant>|<string>|l<label>] , ... ;

Declare the word variables specified by <identifier> (same rules in that labels definition), and optionally define initial values. If the initial value is not defined it's zero.

array byte <identifier>[<word constant>], ... ;

Declare a space for an array with length <word constat> bytes. It's create a label <identifier> with the start array value.

array byte <identifier> = <string>, ... ;

Declare an array of bytes with lenght equal to <string> length with the values of each character in <string> plus zero character ending. It's create a label <identifier> with the start array address value.

array byte <identifier> = { <byte constant>|<character constant>, ... }, ... ;

Declare an array of bytes with length equal to number of elements enclosed in brackets, with the value of each element in list. It's create a label <identifier> with the start array address value.

string <identifier>[<word constant>], ... ;

Equivalent to array byte <identifier>[<word constant> + 1], for better understanding of program. Define an array of bytes of lenght <word constant> + 1, for store a string of maximum <word constant> characters.

string <identifier> = <string>, ... ;

Equivalent to array byte <identifier> = <cadena>], for better understanding of program. Define an array of bytes of lenght <string> + 1, with values of each character of <string> plus zero character ending.

array word <identifier> [<word constant>], ... ;

Declare a space for an array with length length <word constant> words (<word constant> * 2 bytes). It's create a label <identifier> with the start array address value.

array word <identifier> = { <word constant>|<string>|<label>, ... }, ... ;

Declare an array of bytes with length equal to number of elements enclosed in brackets, with the value of each element in list. It's create a label <identifier> with the start array address value.

function [register|inline] byte|word <identifier>(byte|word, ...) { <string>, ... }
[using <function name>, ... ;]

Declare the function <identifier> with the parameters specified in brackets (and) including the assembler instructions <string> defined betweend brackets { and }. The functions are write only in assembler and can load at end the return value in register A (if the function type is byte) o in register pair HL (if the function type is word). The parameters are passed in stack, where parameter #n is in SP + 2 * n + 1, if the argument type is byte, or SP + 2 * n (low byte) and SP + 2 * n + 1 (hight byte), if the argument type is word.

The clause register is valid only for one parameter functions and if specified the value parameter is not passed in stack, but in A register (if parameter is byte) or HL (if parameter is word).

If inline clause is specified the function is specified as an inline function and is not called by CALL instruction, the code function is inserted directly (same as code in an asm sentence). An inline function must have none or one argument, can't use labels and can't be specified in an using clause in another function. If the argument is a byte value, this is loaded in A register before insert the function code; if the argument is a word value, this is loaded in HL register. The function result must be at end of code in A register if the function returns a byte value, and in HL register if the function returns a word value.

If using clause is specified the <function name> in list are associated to declared function. So, if you use the declarad function in the program the function code will add to code object, and also the code function of functions associated, althought if the functions associated are not used in program. This is usefull for use a function code in another function.

if (<expression>) { <sentences 1> }|<sentence 1>;
[else { <sentences 2> }|<sentence 2>;]

Evaluates <expression> and executes <sentences 1> or <sentence 1> if true (not zero) or <sentences 2> or <sentence 2> if false (zero), if specified, or none if not specified.

do { <sentences> }|<sentence>; while (<expression>);

Executes <sentences> or <sentence> while <expression> is true (not zero), evaluating it after each loop.

while (<expression>) { <sentences> }|<sentence>;

Executes <sentences> or <sentence> while <expression> is true (not zero), evaluating it before each loop.

for (<expression 1>, ... ; <expression 2> ; <expression 3>, ...)
{ <sentences> }|<sentence>;

Evaluates <expression 1> list and while <expression 2> is true (not zero) execute <senteces> or <sentence>, evaluating <expression 3> list before each loop.

repeat (<expression>) { <sentences> }|<sentence>;

Executes <senteces> or <sentence> a number of <expression> times.

To exit exit from a repeat loop you need remove the counter value from stack with an asm { “pop hl” } instruction by example.

goto <label>;

Jumps the program control to the address specified by <label>.

gosub <label>;

Jump the program control to the address specified by <label>, until the return sentence, when the program control return to the next sentece to gosub.

return;

Returns the program control to the next sentece to the last gosub executed, or ends the program if not exists previous gosub.

<identifier>:

Declares the label <identifier> with the value of program address where is defined.

<expression>;

Evaluates the <expression>.

Language extend

It's possible if you write libraries of functions, generic or specific for each computer model includig Z80 processor, using the particular ROM, firmware and special characteristics in each computer.

Following the indications in function statement, by example, a function receiving two parameters, the first is a byte and the second is a word, returning a word as the addition of the two parameters, can write it as:

```
function word add_byte_and_word(byte, word)
{
  "ld ix,2",
  "add ix,sp ; IX is pointer to last parameter",
  "ld h,0",
  "ld l,(ix+3) ; HL = 1st parameter (it's a byte and the value is in
                  hight byte of value in stack)",
  "ld d,(ix+1)",
  "ld e,(ix+0) ; DE = 2nd parameter",
  "add hl,de ; HL = 1st parameter + 2nd parameter",
  "ret ; Result is en HL because return function is word"
}
```

An example for an inline function:

```
function inline word neg_word(word)
// Changes the sign of parameter value (returns 0 - parameter)
{
   "; The value parameter is in HL previous to entry function code",
   "ld de,0",
   "ex de,hl",
   "or a",
   "sbc hl,de",
   "; The return value must be in HL at exit function",
   "; The RET instruction is unnecesary because this code",
   "; is not invoked with CALL instruction"
}
```

The same neg_word function without inline clause can be:

```
function word neg_word(word)
{
   "ld ix,2",
   "add ix,sp",
   "ld h,(ix+1)",
   "ld l,(ix+0) ; HL = 1st parameter value",
   "ld de,0",
   "ex de,hl",
   "or a",
   "sbc hl,de ; Result must be in HL at exit function",
   "ret ; RET instruction is necessaty in a not inline function"
}
```

Assembler syntax

The syntax for write the assembler code in functions and in asm statements is based in the GENA assembler of DEVPAC pack. A resume of the main rules for the ccz80 integrated assemblere are:

- Accepts all Z80 instructions including undocumented instructions with registers IXh, IXl, IYh, IYl and the undocumented rotate/shift instructions.

- Labels declaration and symbol definitions with EQU needs : after the name.
- The labels and symbols lenghts is ilimited.
- Constant numbers can be in decimal, hexadecimal (with # prefix) and binary format (with % prefix).
- Constant string and constant characters are delimited by double quotes. Don't use escape sequeces in assembler strings or character constants.
- The expressions can use constants number, constans char, symbols or labels and the operators + (add), - (subtract), * (product), / (integer division), ? (module), & (logic and), @ (logic or) and ! (logic xor). The $ symbol specifies the current counter program value.
- Directives allowed are ORG, EQU, DEFB, DEFW, DEFM, DEFS.
- Case for instructions, register names, labels and symbols is insensitive.
- Conditional directives IF, ELSE, END are not allowed.
- Assembler commands (*E, *H, *S, ...) are not allowed.
- Of course, the number lines are not allowed.

Standard library

The standard library functions are in standard.ccz80 file. You must write the sentece before use any function from this library (usually at begin of source code):

```
include "standard.ccz80"
```

at program start for use it. These functions are:

String manupation functions:

- strlen(<string>): Returns the lenght of <string>
- strcpy(<string 1>, <string 2>): Copy <string 2> in <string 1>. Returns <string 1> address.
- strncpy(<string 1>, <string 2>, <n>): Copy the first <n> characters from <string 2>. Returns <string 1> address.
- strcat(<string 1>, <string 2>): Adds <string 2> to end of <string 1>. Return <string 1> address.
- strset(<string>, <character>, <n>): Sets <string> as <n> times the <character>. Returns <string> address.
- strltrm(<string>, <character>): Deletes <character> at <string> start.
- strrtrm(<string>, <character>): Deletes<character> at <string> end.
- strlpad(<string>, <n>, <character>): Pads at <string> start with <character> to lenght <n>.
- strrpad(<string>, <n>, <character>): Pads at <string> end with <character> to lenght <n>.
- strupr(<string>): Converts to upper case all characters in <string>. Returns the <string> address.
- strlwr(<string>): Converts to lower case all characters in <string>. Returns the <string> address.

- strcmp(<string 1>, <string 2>): Returns 0 if <string 1> = <string 2>, 1 if <string 1> < <string 2> or 2 if <string 1> > <string 2>.

- strchr(<string>, <character>): Returns the memory address where <character> is include in <string> or returns 0 if it isn't include.

- strstr(<string 1>, <string 2>): Returns the memory address where <string 2> is include in <string 1> or returns 0 if it isn't include.

Conversion functions:

- btoa(<string>, <value>): Generates in <string> the decimal representation of <value>. Returns <string> address.

- btoh(<string>, <value>): Generates in <string> the hexadecimal representation of <value>. Returns <string> address.

- bton(<string>, <value>): Generates in <string> the binary representation of <value>. Returns <string> address.

- wtoa(<string>, <value>): Generates in <string> the decimal representation of <value>. Returns <string> address.

- wtoh(<string>, <value>): Generates in <string> the hexadecimal representation of <value>. Returns <string> address.

- wton(<string>, <value>): Generates in <string> the binary representation of <value>. Returns <string> address.

- atob(<string>): Returns the value of <string> as decimal representation.

- htob(<string>): Returns the value of <string> as hexadecimal representation.

- ntob(<string>): Returns the value of <string> as binary representation.

- atow(<string>): Returns the value of <string> as decimal representation.

- htow(<string>): Returns the value of <string> as hexadecimal representation.

- ntow(<string>): Returns the value of <string> as binary representation.

Character functions:

- toupper(<character>: Returns the <character> in upper case.

- tolower(<character>: Returns the <character> in lower case.

Random numbers functions:

- srand(<value>): Defines <value> as the random number seed.

- rand(): Returns a random number between 0 and 65535.

Input/output ports functions:

- in(<port>): Returns the value readed from <port>.
- out(<port>, <value>): Sends <value> to <port>.

Other functions:

- exitrepeat(<address>): Jump to <address> from inside a repeat loop.

Language examples

Example 1. Calculates the sum of the numbers 1 to 100:

```
word sum = 0; // Result of sum
byte i = 1;

while (i <= 100) sum += i++;
printw(s); // printw must be a function for print a word value in
              screen (diferent for each computer: Spectrum, Amstrad
              CPC, ...)
return;
```

Example 2. Finds the maximum value in a word array:

```
array word list = { 50, 225, 32, 450, 1003, 32, 9, 8, #FFFF };
                  // Value #FFFF is the array end mark
word p; // Pointer for each element in array
word maximum = 0;

for (p = list; **p != #FFFF; p += 2) // p is pointer to word, so it's
                                        incremented 2 positions for
                                        access next value in array
  if (maximum == 0) maximum = **p;
  else if (**p > maximum) maximum = **p;

printw(maximum);
return;
```

Example 3. Evaluates some products and sum the results:

```
byte a, b, result = 0;

a = 5;
b = 3;
gosub CalculateProduct; // 5 * 3

a = 1;
b = 4;
gosub CalculateProduct; // 1 * 4

a = 7;
++b ;
gosub CalculateProduct; // 7 * 5

printw(result); // result = 15 + 4 + 35 = 54
return;

CalculateProduct:
result += a * b;
return;
```