The ZTH1 Forth cross-compiler, v. 1.0

S. Morel, Zthorus-Labs, 2022-04-08

1 Introduction

This document describes the Forth language cross-compiler for the ZTH1 CPU. It runs on a Linux/Unix platform. It can be used to develop code for the ZTH1-based computer by generating `.mif` files (memory contents) that can be uploaded using Intel Quartus-Prime.
The fact that the ZTH1 CPU uses a stack of registers makes the Forth language optimal for software development, as Forth is based on the handling of a stack of numerical values.

2 Installation

The ZTH1 Forth cross-compiler consists of a C source file: `zth1f.c`. Once copied into the wished directory of a Linux/Unix computer, it can be compiled by:

```
gcc zth1f.c -o zth1f
```

The ZTH1 Forth cross-compiler also requires the files `os_rom.mif`, `os_ram_h.mif` and `os_ram_l.mif` to be present in the same directory as `zth1a`. These files contain a base OS for the ZTH1 computer that will be integrated into the generated `mif` file.

3 Usage

Source code in Forth for the ZTH1 can be written with any editor, like `vi` or `emacs`. The suffix `.fth` can be used for these source files. The compiller is launched by typing:

```
./zth1f <source file>
```

For example (if the source file is located in the same directory as the compiler):

```
./zth1f my_code.fth
```

If the processing of the source file has been successful, the object files `rom.mif`, `ram_h.mif` and `ram_l.mif` are generated. These files correspond respectively to the memory contents of the instruction ROM, the high-byte data RAM bank and the low-byte data RAM bank of the ZTH1 computer. They can be copied into the Intel Quartus-Prime project of the ZTH1 computer (and then be uploaded into the FPGA implementation of the ZTH1).
An example of ZTH1 Forth program is an implementation for the ZTH1 computer of the Donkey-Kong arcade game which can be found at: `github.com/zthorus/ZTH1-Donkey-Kong`

4 Caveat and limitations

In its current version, this Forth cross-compiler does not strictly follow the standards of other Forth compilers. It has been designed for the ZTH1 architecture as an efficient alternative to the assembly language and has the following limitations and differences, compared to other Forth compilers and interpreters:

1. Variables are defined with a non-standard syntax and it is not possible to define the initial value of a variable from the top of the stack.
2. Variables can only be signed or unsigned 16-bit integers. No 8-bit-integer or floating-point variables are supported.
3. Data are defined with a non-standard syntax.
4. Some new structuring words for branching (i.e., variations of `if` and `until`) have been introduced to optimize the ZTH1 object code.
5. Inclusion of source files from within a source-file is possible.
6. The source code must include the definition of a word called `main` (like in C language) which is executed when the ZTH1 computer is booted.
7. Special care must be taken when using the stack for arithmetic operations: as the Forth stack is actually the ZTH1 CPU register stacks, there are only 8 levels available.
8. Likewise, nested calls to sub-routines are possible only to the 8th level. Considering the use by the ZTH1 OS, user-defined words shall not exceed nested calls to the 4th level. Recursive algorithms are not allowed by the compiler (i.e., the definition of a word cannot contain a call to the word itself).

5 ZTH1 Forth syntax

5.1 General structure

The code written in a `.fth` source file consists of "atoms". Atoms are groups of characters having ASCII codes (in decimal) between 33 and 126 (included). Atoms can be integer values (written in decimal or hexadecimal), Forth words (base or user-defined), meta-characters, parts of character strings, file names. Atoms can be separated by any number of formatting characters like spaces, tabs, carriage-returns.

Comments in the source file must be framed by the atoms '`(`' and '`)`' (standard Forth syntax). In the following code excerpts, comments have to be replaced by the actual code.

The minimal content of a `.fth` file would look like:

```
: main
  ( definition of the main word )
;
```

Likewise, the other words (that are called by the `main` word) are defined by the atom '`:`' followed by the name of the new word, then by the code (i.e., a list of operation on the stack) and terminated by the atom '`;`'. In the same source file, the `main` word can be defined after or before the other

words. The ZTH1 Forth compiler is case-insensitive for the words, the variables, the constants and the hexadecimal values.

It is possible to include from within a `.fth` file the content of other `.fth` files with the sequence of atoms:

```
# ( list of file names separated by space characters ) ;
```

For example, specific user-defined libraries of Forth words can be included by:

```
# math_lib.fth graph_lib.fth string_lib.fth ;
```

5.2 Numerical values

By default, numerical values are decimal on 16 bits. They can be either positive or negative. It is possible to use a numerical value in hexadecimal by using the prefix '`$`'.

Within the definition of a word, and unless it is within the definition of a local constant, any atom consisting of a numerical value means that this value has to be pushed into the top of the stack.

5.3 Variables

Variables are defined by:

```
variable ( name of the variable )
```

The following is also valid:

```
var ( name of the variable )
```

Unlike standard Forth, and because the ZTH1 Forth is compiled. it is not possible to give to a variable, when it is declared, the current value of the top of the stack as default value.

All variables are 16-bit words. They can represent a numerical value or an address in data RAM of a value (i.e., a pointer). It is therefore possible to create arrays of any dimensions (by combinations of the '`@`' and '`+`' Forth words.

Variables can be either global are local. A global variable is defined outside the definition of a word and can be used by any word within its definition. A local variable is defined inside the definition of a word (normally, at the beginning) and can only be used by this word.

In the following excerpt of code,

```
: word1
  var foo
  ( list of atoms defining word1 )
;

: word2
  var foo
  ( list of atoms defining word2 )
  word1
  ( rest of list of atoms defining word 2 )
;
```

it is not a problem for the compiler that `word2` calls `word1` if both words use local variables with the same name. Any action on `foo` during the execution of `word1` will not affect the value of `foo` in `word2`.

5.4 Constants

Constants are defined by:

```
constant ( name of the constant ) ( value of the constant )
```

The following is also valid:

```
cst ( name of the constant ) ( value of the constant )
```

Unlike standard Forth, and because the ZTH1 Forth is compiled. it is not possible to give to a constant, when it is declared, the current value of the top of the stack.

All constants are 16-bit words. They can represent a numerical value or an address in data RAM of a value (i.e., they can represent a variable for which memory space has already been allocated by the system. Hence a constant can be used as a system variable).

Constants can be either global are local. A global constant is defined outside the definition of a word and can be used by any word within its definition. A local constant is defined inside the definition of a word (normally, at the beginning) and can only be used by this word.

In the following excerpt of code,

```
: word1
  cst foo 1000
  ( list of atoms defining word1 )
;

: word2
  cst foo $abcd
  ( list of atoms defining word2 )
  word1
  ( rest of list of atoms defining word 2 )
;
```

it is not a problem for the compiler that `word2` calls `word1` if both words use local constants with the same name. Any action on `foo` during the execution of `word1` will not affect the value of `foo` in `word2`.

5.5 Data

In the ZTH1 CPU, data are stored in a different memory space than the one used for the instructions and are addressed separately (Harvard architecture). In the ZTH1 Forth, data are defined, outside any definition of a word, by the sequence of atoms:

```
data ( address ) ( set of values )
```

The address can be either a 16-bit word, in decimal or in hexadecimal (using the '`$`' prefix), or any atom considered as a "label". In the later case, the compiler will automatically give an address to the declared label (which can be used as a global variable in the word definitions) and will allocate the memory space needed for the set of values.

The set of values can be either a 16-bit word in decimal or a sequence of 16-bit words in hexadecimal that are not separated by space characters and which uses the '`$`' prefix. For example:

```
data array1 $ffffc0001846
```

The set of values can also be a character string (characters are alternatively stored in high-bytes and low-bytes of the 16-bit memory words) by using the '`"`' prefix. For example:

```
data string1 "ABCCDEF
```

No ending '`"`' is needed. Space characters are not allowed in strings defined this way.
It is also possible, instead of defining a set of values, to just allocate a block of memory, by specifying its size (in number of 16-bit words) written in decimal with the prefix '`*`'. For example:

```
data array2 *25
```

6 Base ZTH1-Forth dictionary

In the following table, the Forth stack is described by the stack of ZTH1 registers $(A, B, C...)$, where A is the top of the stack. The programmer shall consider that all the register values used as arguments by a Forth word are dropped after the execution of this word.

6.1 Standard Forth words and meta-characters

`!`	Write value of B at the address specified by A. Do an implicit double `drop` afterwards.
`$`*nnnn*	$(A, B, C...) \Rightarrow (nnnn, A, B, C...)$. *nnnn* is hexadecimal.
`(`	Start a comment. Next atoms will be ignored by the compiler until '`)`' is met.
`)`	Terminate a comment.
`*`	$(A, B, C, ...) \Rightarrow (A * B, C, ...)$. Signed multiplication. A and B shall be between -128 and +127.
`+`	$(A, B, C, ...) \Rightarrow (A + B, C, ...)$.
`-`	$(A, B, C, ...) \Rightarrow (A - B, C, ...)$.
`.`	Display (in decimal) value of A. Do an implicit `drop` afterwards.
`."`	Display a character string defined by the next atoms (until the character '`"`' is met at the end of an atom). Space characters right after '`."`' will be ignored until a non-space character is met and then all space characters are considered until the end of the string.
`/`	$(A, B, C, ...) \Rightarrow (A/B, C, ...)$. Quotient of Euclidian division (A and B are considered as unsigned integers).
`/mod`	$(A, B, C, ...) \Rightarrow (A \ mod \ B, C, ...)$. Remainder of Euclidian division (A and B are considered as unsigned integers).
`1+`	$(A, B, ...) \Rightarrow (A + 1, B, ...)$.
`1-`	$(A, B, ...) \Rightarrow (A - 1, B, ...)$.
`:`	Start the definition of a word (the next atom is the name of the word).
`;`	Terminate the definition of a word.
`<`	$(A, B, C, ...) \Rightarrow (1, C, ...)$ if $A < B$; $(0, C, ...)$ otherwise.
`=`	$(A, B, C, ...) \Rightarrow (1, C, ...)$ if $A = B$; $(0, C, ...)$ otherwise.
`>`	$(A, B, C, ...) \Rightarrow (1, C, ...)$ if $A > B$; $(0, C, ...)$ otherwise.
`@`	$A \leftarrow$ content of address specified by A.

`and`	$(A, B, C, ...) \Rightarrow (A\&B, C, ...)$.
`at`	Set screen location to display text at $(x = A, y = B)$. $(0, 0)$ is the top-left corner. Do an implicit double `drop` afterwards.
`begin`	Start of `begin`/`until` loop.
`cls`	Erase screen.
`cr`	Move to next line to display text.
`do`	Start of `do`/`loop` with first value of iteration = A and last value = B. Do an implicit double `drop` afterwards.
`drop`	$(A, B, C, ...) \Rightarrow (B, C, ...)$.
`dup`	$(A, B, ...) \Rightarrow (A, A, B, ...)$.
`else`	Execute next words (until `then` is met) when the condition of the previous `if` is not met.
`emit`	Display character for which A = ASCII code. Do an implicit `drop` afterwards.
`i`	$(A, B, C, ...) \Rightarrow (i, A, B, C, ...)$. i= current value of iteration inside a `do`/`loop`.
`if`	Execute next words (until `then` or `else` is met) if $A = 1$.
`loop`	Repeat execution of words after the precious `do` if the maximum value of iteration is not reached.
`neg`	$A \leftarrow -A$.
`not`	Toggle all the bits of A.
`or`	$(A, B, C, ...) \Rightarrow (A\|B, C, ...)$.
`over`	$(A, B, C, ...) \Rightarrow (B, A, B, C, ...)$.
`swap`	$(A, B, C, ...) \Rightarrow (B, A, C, ...)$.
`then`	Terminate an `if` or an `if`/`else` sequence.
`u*`	$(A, B, C, ...) \Rightarrow (A * B, C, ...)$. Unsigned multiplication. A and B shall be less than 256.
`until`	Repeat execution of words after the previous `begin` if $A \neq 1$. Do an implicit `drop` afterwards (whatever the value of A).
`xor`	$(A, B, C, ...) \Rightarrow (A\hat{}B, C, ...)$.

6.2 Non-standard Forth words and meta-characters

`#`	Compile source files specified by the next atoms until `;` is met. Source file names cannot include space characters.
`2*`	$A \leftarrow 2 * A$. A is considered as an unsigned integer.
`2/`	$A \leftarrow A/2$. A is considered as an unsigned integer and is rounded down if odd.
`colorsprite`	Set color of sprite A to B (according to the OS color look-up table).
`constant`	Define a global or local constant (the name of this constant will be the next atom, and its value will be the atom after).
`cst`	Same as `constant`.
`defsprite`	Define for sprite A the bitmap (made of four 16-bit words) defined at address B. Do an implicit double `drop` afterwards.
`hidesprite`	Hide sprite A. Do an impicit `drop` afterwards.
`if!=`	Execute next words (until `then` or `else` is met) if $A \neq B$.
`if<`	Execute next words (until `then` or `else` is met) if $A < B$.
`if<=`	Execute next words (until `then` or `else` is met) if $A \leq B$.
`if=`	Execute next words (until `then` or `else` is met) if $A = B$.
`if>`	Execute next words (until `then` or `else` is met) if $A > B$.
`if>=`	Execute next words (until `then` or `else` is met) if $A \geq B$.
`joystick`	$(A, B, C, ...) \Rightarrow (j, A, B, C, ...)$ j=state of joystick=$8 * down + 4 * up + 2 * left + right$. Direction bits are active at 0.
`putsprite`	Display sprite A at $(x = B, y = C)$. $(0, 0)$ is the top-left corner. Do an implicit triple `drop` afterwards.
`rolldn3`	$(A, B, C, ...) \Rightarrow (C, A, B, ...)$.
`rolldn4`	$(A, B, C, D, ...) \Rightarrow (D, A, B, C, ...)$.
`rollup3`	$(A, B, C, ...) \Rightarrow (B, C, A, ...)$.
`rollup4`	$(A, B, C, D, ...) \Rightarrow (B, C, D, A, ...)$.
`until!=`	Repeat execution of words after the previous `repeat` if the condition $A \neq B$ is not met.
`until<`	Repeat execution of words after the previous `repeat` if the condition $A < B$ is not met.
`until<=`	Repeat execution of words after the previous `repeat` if the condition $A \leq B$ is not met.
`until=`	Repeat execution of words after the previous `repeat` if the condition $A = B$ is not met.
`until>`	Repeat execution of words after the previous `repeat` if the condition $A > B$ is not met.
`until>=`	Repeat execution of words after the previous `repeat` if the condition $A \geq B$ is not met.
`var`	Declare a global or local variable (the name of this variable will be the next atom).
`variable`	Same as `var`.

7 Notes on the ZTH1 base OS

The base OS for the ZTH1 computer consists of routines (coded in ZTH1 assembly language) that are called by the following ZTH1 base Forth words: '`*`', '`/`', `/mod`, '`.`' '`."`', `at`, `cls`, `cr`, `emit`, `u*`, `colorsprite`, `defsprite`, `putsprite`, `hidesprite`, `joystick`. It also contains the bitmap of the character set to display text.

7.1 Colors

The color look-up table of the ZTH1 OS is the following:

Value	Color
0	Black
1	Dark blue
2	Yellow
3	Red
4	Pink
5	Cyan
6	Orange
7	Bright blue
8	Grey
9	White
10	Purple
11	Green

Setting the color of a sprite is done by using the Forth word `colorsprite` with the top of the stack containing the sprite number and the second level of the stack contaning the value of the color (as indicated by the table above).

To set the color of text (displayed by the '`.`', '`."`' and `emit` Forth words), the color value has to be written at the address $1873. The high-byte is the color of the characters and the low-byte is the color of the background.

7.2 Redefining the character set

Any character can be redefined (to display graphic elements) by:

```
data ( address ) ( bitmap )
```

The address shall be: $1E80 + 4×(ASCII code of character - 32)

The bitmap consists of a lump of eight 8-bit hexadecimal values.

___oOo___