The ZTH1 assembler, v. 1.0

S. Morel, Zthorus-Labs, 2019-12-31

1 Introduction

This document describes the assembly language tool for the ZTH1 CPU. It is actually a cross-assembler running 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.

2 Installation

The ZTH1 assembler consists of a single C source file: `zth1a.c`. Once copied into the wished directory of a Linux/Unix computer, it can be compiled by:

```
gcc zth1a.c -o zth1a
```

3 Usage

An assembly language source code for the ZTH1 can be written with any editor, like `vi` or `emacs`. The suffix `.asm` can be used for these source files. The assembler is launched by typing:

```
./zth1a <source file>
```

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

```
./zth1a my_code.asm
```

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 be uploaded into the FPGA implementation of the ZTH1).
The assembler takes care automatically of the NOP-padding (see ZTH1 CPU manual) when necessary. At the end of the processing the efficiency of the object code (ratio of the useful code over the used memory) is displayed and gives an idea on how much the NOP-padding has been used.

4 Assembly language rules

Any `.asm` source file shall respect the following rules:

1. Numbers shall be prefixed by the character `&`, `$`, or `x` if the radix is hexadecimal. Numbers using decimal radix shall be prefixed by `d`.

2. Space characters are ignored (except when character string constants are declared, see section xx).

3. It is possible to include comments by using the character `!` followed by the comment.

4. Instructions can be written on the same line if they are separated by the character `;`.

5. The source file shall start (after a possible header in comments) by the instruction `org` followed by an address (in hexadecimal or decimal). `org` indicates the origin address of the instructions written right after it. It is possible to re-define the `org` address several times in the source file. An example of `org` declaration in a source file is:

```
org x0100
```

5 Basic instruction set

The three-character mnemonics for op-codes, as described in the ZTH1 CPU manual, can be used as instructions in the source file, some of them requiring an 8-bit argument that can be typed in the hexadecimal or the decimal radix.

Note: although the ZTH1 CPU manual describes the mnemonics in uppercase characters, the assembler requires to have them written in lowercase characters (case-insensitivity will be a feature of a future version)

6 Macros

To ease the development of code for the ZTH1, built-in "macros" can be used in the source file. Macros are sequences of basic instructions. They are named by four characters (in lower case) and most of them require a 16-bit argument (which can be written in hexadecimal or decimal radix, or be a label, see next section). The available macros are:

Macro	**Function**	**Executed sequence**
`entr` *xxyy*	Shift stack down, set `A` to constant *xxyy*	PSH *xx* ; LDL *yy*
`geth` *xxyy*	Shift stack down, set `AH` to value stored at address *xxyy*	PSH *xx* ; LDL *yy* ; GTH
`getl` *xxyy*	Shift stack down, set `AL` to value stored at address *xxyy*	PSH *xx* ; LDL *yy* ; GTL
`getw` *xxyy*	Shift stack down, set `A` to value stored at address *xxyy*	PSH *xx* ; LDL *yy* ; GTW
`comp` *xxyy*	Set `Z` to 1 if `A`=*xxyy*, set `CF` to 1 if `A`≥*xxyy*	PSH *xx* ; LDL *yy* ; CMP ; DRP
`jump` *xxyy*	Go to instruction at address *xxyy* (set `PC` to *xxyy*)	PSH *xx* ; LDL *yy* ; JMP
`jmpz` *xxyy*	Do a `jump` if `Z`=1	PSH *xx* ; LDL *yy* ; JPZ
`jpnz` *xxyy*	Do a `jump` if `Z`=0	PSH *xx* ; LDL *yy* ; JNZ
`jmpc` *xxyy*	Do a `jump` if `CF`=1	PSH *xx* ; LDL *yy* ; JPC
`jpnc` *xxyy*	Do a `jump` if `CF`=0	PSH *xx* ; LDL *yy* ; JNC

Macro	Function	Executed sequence
`call` *xxyy*	Go to sub-routine at address *xxyy* (shift PS-stack down,set `PC` to *xxyy*)	PSH *xx* ; LDL *yy* ; CAL
`calz` *xxyy*	Do a `call` if `Z`=1	PSH *xx* ; LDL *yy* ; CLZ
`clnz` *xxyy*	Do a `call` if `Z`=0	PSH *xx* ; LDL *yy* ; CNZ
`calc` *xxyy*	Do a `call` if `CF`=1	PSH *xx* ; LDL *yy* ; CLC
`clnc` *xxyy*	Do a `call` if `CF`=0	PSH *xx* ; LDL *yy* ; CNC
`push`	Put `A` into RAM stack	PU1 ; PU2
`popp`	Get `A` from RAM stack	PO1 ; PO2

Note: *xxyy* represents a 16-bit number in hexadecimal. *xx* is the high-byte part and *yy* is the low-byte part.

7 Labels

Labels are character strings starting with the character `@` that represent 16-bit constants. Labels can only be used in macros as 16-bit arguments.
There are two ways to declare labels, depending on the purpose:

1. Declaring a label as a target address for a `jump`, `jmpz`. `jpnz`, `jmpc`, `jpnc`, `call`, `calz`, `clnz`, `calc` or `clnc` macro. In that case, just fill a line in the source file with the label name. For example:

   ```
   @loop1
   ```

   which can be used in the code by an instruction like:

   ```
   jpnz @loop1
   ```

2. Declaring a label as a constant to be used by `entr`, `geth`, `getl`, `getw` or `comp` macros. In that case, fill a line in the source file:

   ```
   @ptr_variable_x=&1A23
   ```

   The labels can be defined anywhere in the source code, even after they are used by a macro.

8 Data declaration

It is possible to declare data values that will be used by the source code (constants, initial values of variables). These data will be stored in the RAM of the ZTH1 computer. It is recommended to declare the data at the end of the source file, after all the instructions (basic instructions, macros and labels) have been written. Declared data will be stored from the address defined by an `org` instruction. Declared data shall be prefixed by the `#` character. Possible data declarations are:

1. 16-bit numbers in hexadecimal. For example:

   ```
   #x1FE0
   #$AB86
   #&6CD2
   ```

2. 16-bit numbers in decimal. For example:

   ```
   #d65535
   ```

3. ASCII character strings, framed by " characters. For example:

   ```
   #"Hello World !"
   ```

___oOo___