# An Assembler for the Tiny Computer

Chuck Thacker, MSR

16 November, 2009

## Introduction

The program TCasm.exe is an extremely simple assembler for the Tiny Computer. To invoke it from a shell, type: “TCasm source memName”. memName is the names given to the three output files produced by the assembler. It is postpended with 0 for the instruction memory, 1 for the data memory, and 2 for the register file. Although it is possible to specify the contents of all three memories this way, only the instruction memory file (memName0.coe) is important.

After assembly, the program is “loaded” by regenerating the instruction memory core in the design using the ISE. This preloads the bitstream file with the contents of the instruction memory. The register file and the data memory are initialized to zeros when the FPGA is configured.

## The assembler

The assembler is very simple, doing most of its work by looking up textual tokens from the source in a symbol table, and emitting the instruction fragment into the current instruction. The symbol table is populated with definition directives placed at the start of the source file, and by symbols representing labeled statements when these are encountered during assembly. An instruction must occupy a single line of source text, and must be terminated with “;”. Any remaining text on the line is ignored. Case is significant. Numbers may be decimal or hex (0xnnn). Tokens are separated by white space.

The assembler is two-pass. During the first pass, the assembler skips tokens that are undefined. Isolated strings that resolve to fields are placed in currentValue with the indicated offset. Isolated numbers are also placed in currentValue, using an offset of 3.

Symbol table entries have a *key* and *contents* which is a *field* containing a *value* and an *offset* in the instruction (the bit position into which the value should be placed). Other built-in reserved words are treated as follows:

“field *string value offset*” declares a symbol. The next token must be a string and the next two must be numbers (or names that resolve to numbers). This creates a symbol with *key* = *string* and *contents* = {*value, offset*}.

"rfref *string number*" defines three symbols, with keys that are concatenations of "a", "b and "w" with the *string*, The contents' values are *number*, and their offsets are aoff, boff, and woff respectively. These constants must be defined as fields before the *rfref*. For the TC, aoff = 22, boff = 10, and woff = 27. The resulting symbols are used in later instructions to name the register file's a, b, and w instruction fields for register *number*.

"mem string (which must evaluate to a number)" or "mem *number*" makes M[number] the current memory. Usually the "mem string" form will be used, after a preceding *field* definition of the memory name. Memories are numbered 0 to 2. Token processing continues after the string or number.

"loc *string* (which must evaluate to a number)" or "loc *number*" sets the current location in the current memory.

"string:" Provides a way to label statements. A symbol is defined with *key* = ("L" + string), *contents* = {currentLocation, 10}. This is an Rb constant. Execution of a "Jump Lloc " instruction substitutes this (12 bit) constant for RF[Rb] as the "b" input to the ALU, providing that location "loc" has been labeled. If Ra = 0, the Jump is absolute, since r0 contains zero.

It is also possible to do PC- relative Jumps: If Ra = 31, PC is used as the "a" input to the ALU instead of RF[Ra]. If both "*rfref* PC, 31" and "*rfref* Offset *displacement*" have been previously encountered in the source, then "Jump aPC + bOffset Jumps to PC + *displacement.*

When token processing for a line is finished, if any field in currentValue has been set, the value is placed into the current location in the current memory, and the current location in that memory is incremented. currentValue is cleared and scanning resumes at the start of the next line.