A Universal Assembler for the Tiny Computer

Chuck Thacker, MSR
11 September, 2007

Introduction

The “Tiny Computer” is likely to become part of the BEE3 infrastructure. Unfortunately, we need to write programs for each instantiation of it, in the several versions of the processor that we currently plan to build.

These aren’t complicated programs. They only need to provide initialization files for the three block RAMs used in the TC: the data memory, the instruction memory, and the register file. The format of an instruction may change from version to version, and it would be nice to have an assembler that was insensitive to these changes.

The assembler described here meets this goal. It is quite dumb – it has very little syntax, and does most of its work by doing lookups in a symbol table. The program text carries both the definition of the machine and the specification of the three memories. This will allow us to tinker with code sequences for various versions before committing to a particular instruction format.

The assembler data structures

The assembler data structures are simple. There are three arrays, one for the instruction memory, one for the data memory, and one for the register file. All hold 36-bit values, since this is the maximum width of the data paths in the TC. They are 1K words in length, since this is the size of the memories. Each array has an associated “current location” pointer, which points to the first empty register in the associated memory.

There is also a “currentValue” register that contains the current (assembled) value for the text line being analyzed by the assembler.

There is a token array, containing the tokens parsed from the text stream representing the current line of input. The size of this array is determined by the maximum number of tokens per instruction -- probably about 50.

There is also the symbol table. This maps string keys to values, which are *fields* containing the *value*, and the *offset* (in bits) of the value in the target memory.

Case is significant in strings. Numbers are decimal.

The assembler is line-oriented. A scanner breaks each line into a series of *tokens*, which are numbers, built-in reserved words, or strings. These are stored in the token array, which is then processed. A semicolon causes the remaining characters in the line to be skipped, and positions the scanner at the start of the next line after processing any tokens. When token processing for a line is finished, the token array is cleared, and made ready for the next source line.

The assembler is two-pass, to provide for the resolution of forward references. During the first pass, the location counter(s) are modified as above, but no code is placed in the memories. In the second pass, all symbols should have been resolved (an undefined symbol is an error), and memory contents are generated. When processing of the input is complete, the three memories are written to the files needed for the FPGA configuration bitstream, and the assembler exits.

There are a few built-in reserved words, "*field*", "*mem*", "*loc*", ":", and "*rfref*".

The token processor skips tokens that are undefined during pass 1. 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 0. Other built-in reserved words are treated as follows:

1) "*field*, string, number, number" declares a field. The next token must be a string and the next two must be numbers (or names that resolve to numbers). . Token processing continues at the fourth token after the field.. During pass 2, all symbols should have been defined, so symbol definition is elided.

2) "*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.

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

4) "string:" Provides a way to label statements. A symbol is defined with key = string, value = current location in the current memory, and offset 0. The ":" must only appear as the second token on a line (the first is the string). During pass 2, the string will have been evaluated and (incorrectly) placed in currentValue when the ":" is encountered, so currentValue is cleared.

7) "rfref string number;" defines *three* symbols, with keys that are concatenations of "a", "b and "w" with the string, and offsets of aoff, boff, and woff respectively. These constants must be defined before the *rfref*. For TC3, aoff = 17, boff = 10, and woff = 25.

The resulting symbols are used in later instructions to reference the register file via the a, b, and w ports.

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.

The entire machine description for the TC (version 3) is plus a small program that initializes Rcnt to 10, counts it down to 0, and repeats is shown below:

```
field aoff  17 0;  define the field offsets for rfref. These names must be defined and
must not change.
field boff  10 0;
field woff  25 0;

field instruction 0 0; symbolic name for instruction memmory
field rf 1 0; symbolic name for register file
field data 2 0; symbolic name for data memory

field PC 0 0; noise word
field <= 0 0; noise word
field , 0 0; noise word

field lc 1 24; the "load constant" bit

field +  0  7; the "plus" function
field -  1  7;  the "minus" function
field ++ 2  7; the "Rb + 1" function
field -- 3  7; the "Rb - 1" function
field &  4  7; the "and" function
field |  5  7; the "or" function
field ^  6  7; the "xor" function

field rcy1  1  5;
field rcy8  2  5;
field rcy16 3  5;

field skn 1  3; skip if ALU < 0
field skz 2  3; skip if ALU = 0
field ski 3  3; skip if InRdy

field Store  1 0; Opcodes
field StoreI 2 0;
field Out    3 0;
field Load   4 0;
field In     5 0;
field Jump   6 0;

mem instruction loc 200;  Set current memory to the instruction memory, location 200.
rfref Rcnt 1; A program to load Rcnt with 10, count it down to 0, then jump back to loop
and repeats.
rfref LocLoop 2;
rfref LocDec 3;
rfref Zero 4; zero
rfref Trash 5; trashcan

        wLocLoop <= lc loop;
        wLocDec <= lc dec;
loop:   wRcnt <= lc 10;               initialize Rcnt
dec:    wRcnt <= bRcnt -- skz; decrement Rcnt, skz
        Jump aLocDec | bZero, wTrash <= PC;
        Jump aLocLoop ^ bZero, wTrash <= PC;
end
```

When the assembler is run with the example above, it produces the following console output:

```
Memory 0: 6 location(s) used
 200: Rw = 2, LC= 1, Const = 202
 201: Rw = 3, LC= 1, Const = 203
 202: Rw = 1, LC= 1, Const = 10
 203: Rw = 1, LC= 0, Ra = 0, Rb = 1, F = 3, Sh = 0, Sk = 2, Op = 0
 204: Rw = 5, LC= 0, Ra = 3, Rb = 4, F = 5, Sh = 0, Sk = 0, Op = 6
 205: Rw = 5, LC= 0, Ra = 2, Rb = 4, F = 6, Sh = 0, Sk = 0, Op = 6
Memory 1: 0 location(s) used
Memory 2: 0 location(s) used
```

The syntax (or lack of it) allows a number of possibly strange-looking instructions to be correct syntactically. Noise words (e.g., "instruction", "rf", and "data" in the example), may be added to improve clarity. These are defined as fields with value = 0, offset = 0, so they are essentially ignored. Some common patterns should be adopted to make the source text readable. For the TC3, some examples are:

1) An instruction with an optional label that reads two registers Rx and Ry, and writes the result to Rz, optionally cycling the result right by 8 and optionally skipping the next instruction if the result is zero (optional things are shown in []s):

```
[label:]  wRz <= aRx + bRy, [rcy8] [skz];
```

2) An instruction that stores Rx to the location given by Ry, decrements Ry and writes Ry with the result, skipping if the decremented Ry is zero:

```
[label:]  Store aRx, wRw <= bRy--, skz
```

But this is a fairly silly instruction. More likely, Ry would be tested against a limit in a second instruction, which would skip over a Jump when the limit is reached.

3) An instruction that jumps to the location given by Rx + Ry, writing the incremented PC to Rz:

```
[label:] wRz <= PC, Jump aRx + bRy;
```

4) Load a constant into Rx:

```
[label :]   wRx <= lc 16'hffff';
```

5) Jumping to a labeled instruction means that the location denoted by the label must have previously been placed in RF. The incremented PC is placed in Ry:

```
wRx <= label, lc; Loads a constant pointing to "label" into Rx.
wRy <= PC, Jump aRx + bRzero;
```

Rzero is a register known to hold the constant zero. If “label” is a forward reference, it won’t be defined until pass 1 is finished, so it is simply skipped during pass 1. If the program has only a few labeled instructions, it might be a good idea to preload them into the register file. The example above does exactly this. Having a lot of registers helps here.

6) Fill the data memory with four numbers, starting at location 200.

```
mem data loc 200; switch to data memory
number0;
number1;
number2;
number3;
mem instruction; Switch back to IM, at the previous location.
DM location is now 204.
```

There are other patterns that might make sense. These are just a few possibilities.