# AIK, the Assembler Interpreter from Kentucky

H. G. Dietz and W. R. Dieter

hankd@engr.uky.edu dieter@engr.uky.edu

Electrical & Computer Engineering Department

University of Kentucky

Lexington, KY 40506-0046

5th April 2007

**Abstract**

Building an assembler is not all that hard, but it is not a trivial task and quite frequently it is a necessary evil. Over the past few decades, various "generic" assemblers and assembler generators have been developed to ease the burden. However, these systems still tend to be big and complex enough so that there is a significant learning curve associated with their use. In contrast, AIK is designed to be a simple, self-contained, tool that interpretively assembles according to a specification given in a very straightforward notation.

An *assembler* is a program that converts human-readable machine-level instructions into their machine readable form. Although assemblers have been around for half a century, there are a few issues that still make it a pain to write an assembler – the worst of which is the issue of how to resolve forward references. The idea behind AIK is to make it easy enough to build an assembler so that typical undergraduate students will be able to experiment with their own assembly languages and instruction encodings... without needing any background in compiler construction.

To achieve this goal, the input to Aik is actually in two pieces: the specification of the assembly language and the assembly language program (instructions) you wish to assemble.

## 1 Specifications

The specification of an instruction is disturbingly simple. Let's consider the simple instruction set from page 152 of Tanenbaum's *Structured Computer Organization* textbook. The AIK specification could look like Figure 1. Let's break that description down a bit.

The `SPECIFICATIONS` heading says that this is a list of specifications, as opposed to instructions to be processed as per specifications. Each subsequent line describes one type of instruction, and each type of instruction should be described by only one line.

The portion of each line to the left of the `=` describes the syntax of the instruction. For example, `PUSH addr` is defining the syntax of a "PUSH" instruction as the keyword `PUSH` followed by an arbitrary C-style expression (see Section 2.1) whose value can be referenced by the name `addr`.

```
SPECIFICATIONS
PUSH addr = 0:3 addr:13
POP addr = 1:3 addr:13
JUMP addr = 2:3 addr:13
JNEG addr = 3:3 addr:13
JZER addr = 4:3 addr:13
JPOS addr = 5:3 addr:13
CALL addr = 6:3 addr:13
ADD = 7:3 0:13
SUB = 7:3 1:13
MUL = 7:3 2:13
DIV = 7:3 3:13
RETURN = 7:3 4:13
```

Figure 1: AIK Specification of Tanenbaum's Target Machine

```
SPECIFICATIONS
ONEARG addr = THIS:3 addr:13
= ONEARG PUSH POP JUMP JNEG JZER JPOS CALL
NOARG = 7:3 THIS:13
= NOARG ADD SUB MUL DIV RETURN
```

Figure 2: AIK Specification Using Format Aliases

```
beq $rs,$rt,lab = 4:6 rs:5 rt:5 ((lab-(.+4))/4):16
```

Figure 3: AIK Specification Of MIPS `beq`

More specifically, `addr` is significant only in that it is a name; any name can be used in specifying instruction syntax. Be warned, however, that these names will be treated in a way that can interfere with use of the same names as labels, etc., within a program to be processed. Things that are not names must literally be matched. For example, if we wanted to require brackets around the argument to `PUSH`, we could write `PUSH [addr]`. The recommended symbols for such "syntactic sugar" in these patterns are `$`, `#`, `[`, `]`, `,`, and `@`; other symbols are too easily confused with other elements of the AIK syntax.

The portion of each line to the right of the `=` describes the encoding of the instruction. The encoding is specified by listing the bitfields within the encoded structure in order of descending significance – the high bit positions are listed first. There may be arbitrarily many bitfields in the instruction encoding, each of which consists of an arbitrary C-style expression followed by a colon and a second C-style expression. The first expression is used to compute the value of the field, the second computes the number of bits in the field (which normally would be a constant).

## 1.1 Format Aliases

The specifications shown above are fairly obvious and very expressive, but the basic specification syntax has a fair amount of redundancy across instructions. To reduce redundancy, AIK provides a very simple format alias mechanism.

Revisiting our earlier example, Tanenbaum's instruction set has 12 instructions, but there are really only two instruction formats. What we need is a way to give parameterized descriptions of these two instruction formats. For AIK, this is done by specifying each format is much the same way a single instruction of that type would be described. In Figure 2, we give these two formats the names `ONEARG` and `NOARG`. The parameterization uses the keyword `THIS` in the encoding portion of the pattern to stand for the value associated with each of the possible substitutions (aliases) for the instruction name.

At any point in the specification after a format has been given, a set of aliases for that format can be specified. As shown in Figure 2, aliases are enumerated using an alias specification that begins with `=` and the name of the format. By default, the aliases are given values counting from 0, although this default behavior can be overridden by inserting a constant number before any of the symbols to cause numbering to continue from that value. Thus, in Figure 2, `JUMP` is not just identified as a parameter for the `ONEARG` format, but also is given the value 2, which will be accessible in that format as the value of `THIS`.

## 1.2 Complex Codings

In order to support more complex encoding, expressions can be used in the right side of a specification. For example, the MIPS instruction set has a branch-equals instruction in which the low 16 bits of the instruction are actually stored as the top 16 bits of a signed 18-bit address offset based after the instruction. There also is some quirky syntactic sugar, such as register names being preceded by the `$` symbol. However, the instruction specification is simply as shown in Figure 3, in which the only surprise is the fact that `.` represents the current location counter – a fairly common notation in assembly languages.

Perhaps the most shocking thing is that the field widths in the encodings are permitted to be arbitrary C-style expressions that can have different values in different instances of the same

instruction. Thus, variable-length encodings are supported. To make this more effective, fields with width less than 1 bit are literally omitted from the encoding.

# 2 Instructions

After the specifications comes the code to assemble. The end of the specifications is marked by the keyword `INSTRUCTIONS`. The assembly language input looks essentially as one would expect, with a few differences outlined in the following subsections.

## 2.1 Expressions

As mentioned earlier, AIK expression syntax is largely derived from C.

Constant integer values (which are generally treated as signed) can be expressed as they would be in C using hexadecimal, decimal, or octal. To this, we add support for binary constants. For example, all the following are essentially equivalent: `0x11`, `17`, `021`, and `0b10001`. Currently, integer values are tracked with 32 bit precision.

AIK allows a wide range of C-style operators in expressions, as described in Table 1. Fundamentally, this is the full set of C operators that do not involve assignment. Assignment operations (`++`, `--`, `=`, and *op*`=`) are not supported by AIK.

## 2.2 Built-In Operations

In addition to the instructions that the user specifies, there are a modest number of built-in pseudo operations. The currently supported pseudo operations are:

**`TEXT`** Select the program text (code) segment. The text segment location counter is active.

**`DATA`** Select the data segment. The data segment location counter is active.

**`ORG`** ***expr*** Set the active location counter to the value of the expression *expr*.

**`DW`** ***expr*** Define a word to have the value of the expression *expr*.

**`DS`** ***expr*** Define a space of as many 0 values as the value of the expression *expr*.

***lab*** **`EQU`** ***expr*** Equate the value of the label *lab* to the value of the expression *expr*. Any change in the equated value from the value given in the previous pass will cause an additional pass.

***lab*** **`SET`** ***expr*** Set the value of the label *lab* to the value of the expression *expr*. Unlike `EQU`, changes in the set value will not cause an additional pass.

**`IF`** ***expr*** Conditional assembly support; only generate code for the following assembly source if expression *expr* is not equal to 0. Conditionals may be nested arbitrarily.

**`ELSE`** Conditional assembly support; only generate code for the following assembly source if generation was enabled until the most recent open `IF`. Multiple `ELSE` can be applied within a single open `IF` to flip the condition repeatedly.

| Operator | Level | Group | Meaning |
|---|---|---|---|
| *x* ? *y* : *z* | 0 | | if (*x*!=0) return(*y*) else return(*z*) |
| *x* \|\| *y* | 1 | > | if (*x*!=0) return(1) else return(*y*!=0) |
| *x* && *y* | 2 | > | if (*x*!=0) return(*y*!=0) else return(0) |
| *x* \| *y* | 3 | > | return(bitwiseOR(*x*, *y*)) |
| *x* ^ *y* | 4 | > | return(bitwiseXOR(*x*, *y*)) |
| *x* & *y* | 5 | > | return(bitwiseAND(*x*, *y*)) |
| *x* == *y* | 6 | > | if (*x*==*y*) return(1) else return(0) |
| *x* != *y* | 6 | > | if (*x*!=*y*) return(1) else return(0) |
| *x* < *y* | 7 | > | if (*x*<*y*) return(1) else return(0) |
| *x* <= *y* | 7 | > | if (*x*<=*y*) return(1) else return(0) |
| *x* > *y* | 7 | > | if (*x*>*y*) return(1) else return(0) |
| *x* >= *y* | 7 | > | if (*x*>=*y*) return(1) else return(0) |
| *x* >> *y* | 8 | > | return(signedShiftRight(*x*, *y*)) |
| *x* << *y* | 8 | > | return(signedShiftLeft(*x*, *y*)) |
| *x* + *y* | 9 | > | return(add(*x*, *y*)) |
| *x* - *y* | 9 | > | return(subtract(*x*, *y*)) |
| *x* * *y* | 10 | > | return(multiply(*x*, *y*)) |
| *x* / *y* | 10 | > | return(signedDivide(*x*, *y*)) |
| *x* % *y* | 10 | > | return(signedModulus(*x*, *y*)) |
| ( *x* ) | 11 | | return(*x*) |
| + *x* | 11 | < | return(*x*) |
| - *x* | 11 | < | return(subtract(0, *x*)) |
| ! *x* | 11 | < | if (*x*!=0) return(0) else return(1) |
| ~ *x* | 11 | < | return(bitwiseNOT(*x*)) |

Table 1: AIK Expression Operators

```
beq $rs,$rt,lab = {
    { 4:6 rs:5 rt:5 }   ; a comment
    ((lab-(.+4))/4):16  ; another comment
}
```

Figure 4: Multi-line AIK Specification Of MIPS `beq`

**END** Conditional assembly support; close the most recent open `IF`.

Currently, there is no way for the user to change the names or functionality of these operations. However, it is trivial to construct your own specification of an operation like `DW`.

Conspicuously missing are the fancy macro facilities found in some assemblers, as are the hooks commonly used for interfacing with external linkers. Fundamentally, AIK is intended to serve as a target for compilers or for hand-writing relatively small assembly programs, so fancy macro support would add complexity without significant benefit. The lack of a linker interface also is the direct consequence of a design goal: AIK is intended to target very raw, often custom-designed and FPGA-implemented, hardware systems that don't know what to do with something as sophisticated as an ELF binary. The conditional assembly support in AIK is intended to function as a "poor man's linker" to intelligently integrate library code into an application, still producing pure executable code in any of several very mundane formats.

# 3 Miscellaneous Issues

AIK uses a classical line-oriented lexical structure. Each statement is one line long, with the end of line marking the statement end. Comments begin with a semicolon, `;`, and continue to the end of the line. However, sometimes a line would have to be very long to accommodate an instruction or, even more likely, a specification. Thus, AIK allows end of line markings to be ignored if they appear between `{` and `}`; these braces can appear anywhere between symbols, and also can be nested so that the user may employ them to improve the human readability of a specification by suggesting grouping. A simple example of the use of this multi-line support is given in Figure 4.

Lexical analysis is in most other ways similar to that used for C code. Whitespace has no significance other than potentially serving as an edge between symbols. Names are like C identifiers, consisting of any sequence of one or more letters, digits, or the underscore character, such that the first character is not a digit. Names are case sensitive.

Although AIK only reads its input once, internally it is a full multi-pass assembler that iterates as needed to resolve forward references and span-dependent interlock issues. The passes are made over a tokenized intermediate representation, which makes multi-pass evaluation very swift. However, the number of passes is *not* bounded, and relatively simple errors in the input can result in an infinite loop.

The use of a tokenized intermediate representation also explains why AIK does not produce a pretty assembly listing – or very nice error messages. The current version of AIK literally does not keep the source around to use for such things.

The WWW form-based version of AIK differs from the command line version primarily in that the command line version reads from the standard input and writes to the standard output, whereas the form-based version dynamically generates HTML forms for input and output. AIK takes advantage of the fact that HTML forms easily can embody multiple text areas. The input is separated into two independent text boxes for the specifications and the instructions, neither of which contains the keywords `SPECIFICATIONS` or `INSTRUCTIONS`. The output is similarly split into separate areas within the HTML for the `TEXT` and `DATA` segments.

# 4 Acknowledgements

AIK was created in Spring 2007 by Professor Hank Dietz, who has built many compilers, assemblers, and language tools before. In fact, AIK is constructed using a tool set he helped create many years ago – PCCTS, ANTLR version 1.33MR33. However, the motivation and many of the functionality concepts came from Professor Bill Dieter. In Spring 2007, Bill was for the first time teaching the EE480 Advanced Computer Architecture course at the University of Kentucky. He wanted to have the undergraduate students design their own assembly language and instruction set encoding. However, he also wanted the students to be able to write and run significant assembly language programs on FPGA hardware or FPGA simulation software.