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

15th 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 the first edition of Tannenbaum's *Structured Computer Organization* textbook [7]. The AIK specification could look like Figure 1. Let's break that description down a bit.

This is a list of specifications, as opposed to assembly code to be processed as per specifications. Each subsequent line describes one instruction pattern in which the pattern begins with the name of the instruction. If a particular instruction has several different forms, each form is described with a separate pattern on a separate line. There is a limit on the number of alternative forms permitted for each instruction, currently 8.

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

```
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 Tannenbaum's Target Machine

```
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

```
= 0 zero 2 v0 v1 a0 a1 a2 a3
= 8 t0 t1 t2 t3 t4 t5 t6 t7 24 t8 t9
= 16 s0 s1 s2 s3 s4 s5 s6 s7
= 28 gp sp fp ra
```

Figure 3: AIK Specification Of Values For MIPS Register Names

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

Figure 4: AIK Specification Of MIPS `beq`

```
JBR L ?((L-.<=256)&&(L-.>=-254)) = BR:8 (L-.)/2:8
JBR L = JMP:16 L:16
```

Figure 5: AIK Specification Of Span-Dependent PDP11 `JBR`

arbitrary C-style expression (see Section 2.1) whose value can be referenced by the name `addr`. 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 Instruction 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 instruction format alias mechanism.

Revisiting our earlier example, Tannenbaum'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 in 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. Note that the alias duplicates all alternative formats that exist at the point the alias is created. 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`. Alternatively, borrowing loosely from the patterns syntax used for NEW JERSEY MACHINE-CODE TOOLKIT [4], one can use the name _ to indicate a skipped value; more precisely, AIK implements this behavior by the fact that it does not forbid any symbol from being defined multiple times, so any name can be used as a dummy.

The alias syntax also can be used to efficiently initialize a sequence of names without creating aliases for them. In such a case, a number must be the first thing after the `=` symbol. For example, the values of the symbolic names for the MIPS registers can be specified as shown in Figure 3.

1.2 Complex Encodings

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 4, 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.

Another way to express variable-length encoding takes advantage of the fact that there may be several different patterns for coding the same instruction. In the pattern specification, the `?` symbol may be used to introduce a parenthesized expression anywhere in the left side that, if it evaluates to `0`, will disable application of the current pattern. Szymanski's paper on handling of span-dependent instructions[6] uses the PDP11 `JBR` (`JMP` or `BR`, automatically selected based on span) instruction for its examples; it also serves as a good example here. Figure 5 shows how this instruction can be specified using AIK. This specification takes advantage of the inherent order of attempting to match the two patterns, trying the more general pattern only when the special case has failed. It is useful to note that AIK's multi-pass resolution is much faster than that of a conventional assembler because it only makes multiple passes over a pre-parsed internal form rather than over the assembly source, but it is not as fast as Szymanski's algorithm. On the other hand, the method used in AIK allows instruction encoding to be variable in ways that are far more complex than Szymanski's algorithm can handle. The `?` expressions are permitted to evaluate expressions using any symbols defined up to that point in matching the instruction pattern.

1.3 AIK Option Variables

There are a variety of built-in variables that can be used to control behavior of AIK. Note that all of them have names that start with the character sequence `AIK`. There are two fundamentally different types of option variables. The first type is literally an initialized symbol whose value is numeric; the second type is not really given a numeric value, but instead has a value which is essentially a string. Both types of symbol can be `SET` or `EQU`ated to values within a specification, but only the option variables with a numerical value can be accessed within an assembly language program. Although changing the value of such a variable within an assembly program is permitted, inconsistent behavior may result because option variables are examined when it is convenient, making it unclear when a change suggested by changing an option variable would be implemented. Table 1 lists the option variables currently implemented.

Note that most of the properties of the `TEXT` and `DATA` segments can be set together or can be set independently. Not only can they be different depths and widths, but they also can be output in different formats. This is permitted because it is fairly common that the `TEXT` segment would be

Variable	Default	Use In Programs?	Meaning
`AIKAsmExt`	`asm`	no	Suffix for assembler source files, `.asm`
`AIKTextExt`	`text`	no	Suffix for `TEXT` file, `.text`
`AIKDataExt`	`data`	no	Suffix for `DATA` file, `.data`
`AIKTextDepth`	`65536`	yes	Number of addresses in `TEXT` file
`AIKDataDepth`	`65536`	yes	Number of addresses in `DATA` file
`AIKDepth`	`0`	yes	Number of addresses in each segment
`AIKTextWidth`	`8`	yes	Memory word size for `TEXT` in bits
`AIKDataWidth`	`8`	yes	Memory word size for `DATA` in bits
`AIKWidth`	`0`	yes	Memory word size for both segments
`AIKLowFirst`	`0`	yes	Order multi-word values low-to-high?
`AIKTextFormat`	`VMEM`	no	`TEXT` file record format; choices are: `I8HEX` for Intel 8-bit HEX [2] `SREC` for Motorola S-Records [3] `MIF` for Memory Initialization File [1] `VMEM` for Verilog MEM [5]
`AIKDataFormat`	`VMEM`	no	`DATA` file record format, as above
`AIKFormat`	`VMEM`	no	Set both `TEXT` and `DATA` file formats
`AIKPasses`	`20`	yes	Maximum number of internal passes

Table 1: Option Variables For Use In Specifications

in ROM and `DATA` in RAM, thus using completely different mechanisms to load the contents of the two segments. Setting the depth or width for both segments together is done by changing the value of `AIKWidth` or `AIKDepth`; the new value is copied to the segment-specific settings at the end of reading the specifications.

There are various restrictions implied by some of the options. For example, the `I8HEX` output format does not allow an address to be larger than 16 bits nor a word to be larger than 8 bits. Currently, error reporting for violation of such restrictions is minimal if the error is detected at all.

2 Assembly Language Code

After the specifications comes the code to assemble. 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 2. 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*. This directive also will accept a list of *expr* or one or more quoted strings, any of which will result in a series of defined word values.

`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. It is useful to note that an as yet undefined symbol always has the value 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.

`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.

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 2: AIK Expression Operators

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

Figure 6: Multi-line AIK Specification Of MIPS `beq`

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 6.

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. A label on a statement *must* be followed by a colon (`:`) – the colon is not optional, as is common for many assemblers.

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, relatively simple errors in the input can result in a failure to converge on a stable set of values, with little if any debugging help provided by AIK.

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 and writes files, 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. The output is similarly split into separate areas within the HTML for the `TEXT` and `DATA` segments.

4 Status & Future Work

The motivation and many of the functionality concepts for AIK came from Professor Bill Dieter. In Spring 2007, Bill was for the first time teaching the *EE480 Advanced Computer Architecture* course in the Department of Electrical and Computer Engineering 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.

Thus, 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. AIK is not the first assembler-building tool Dietz has created; in 1981 he created ASA (the ASsembler Adaptable), which assembled code for any of at least 30 different processors. Many of AIK's features came from ASA, including the command line structure. The name ASA was intended to be read "as a" – assembling `t` as 6502 assembly code used a CP/M command line like `asa 6502 t`. Incidentally, we pronounce AIK as the English word “ache” – appropriate enough, assemblers being the dull persistent pain that they are.

The AIK project is and will continue to be an open source effort with as few use restrictions as possible, so feel free to use and modify the source code as you see fit. We do ask that the code and any documents referring to it appropriately cite this paper using something like the BIBTEX entry:

```
@techreport{AIK,
 author={H. G. Dietz and W. R. Dieter},
 title={Aik, the Assembler Interpreter from Kentucky},
 institution={University of Kentucky},
 month={April},
 year={2007},
 howpublished={Aggregate.Org online technical report},
 url={http://aggregate.org/AIK/aik.pdf}
}
```

Of course, the user assumes all responsibility for any problems or other unfortunate side effects that may result from use of AIK, and agrees to hold the authors and the University of Kentucky harmless.

At this writing, AIK is very young – it has only been available to friendly users for a few weeks. Even in that short time, there have been a number of weaknesses identified and repaired. We expect this active user feedback process to continue for some years, and strongly encourage users to send their suggestions to the authors via email. Possible extensions being considered include:

- Improvements to the error handling, which is currently exceedingly poor.
- Possible addition of a facility for interpretive disassembly. It is believed that this can be done using the current syntax, although expressions within the specifications might need to be restricted somewhat. In this respect, the goal would be to provide functionality similar to that provided by the NEW JERSEY MACHINE-CODE TOOLKIT [4], which does not literally build assemblers, but does use a single specification to construct functions that aid in assembly/disassembly of instructions.

It is not our intention that AIK should become a production assembler for any popular computer processor; perhaps we’ll eventually make an assembler compiler (ACK?) for that. The goal is to make it very simple to assemble code for experimental, application-specific, or student-project processors. These have in common that they most often are implemented using VHDL or VERILOG and FPGA hardware, with a minimal runtime software environment. For this reason, we do *not* currently intend to add support for ELF binaries, relocation, symbolic debuggers, etc.

Appendix: ANTLR Grammar For AIK

Figure 7 gives the complete grammar for AIK input. In reality, it is two grammars with many rules in common but different start symbols; `specifications` starts the specification grammar and `instructions` starts the assembly language grammar. This is the actual grammar used to construct the tool, but all actions and C code have been removed to improve readability. This grammar is not even close to being LL(1), but ANTLR handles it efficiently. There is a minor ambiguity when two expressions may appear in sequence and the second one has a unary sign; unless the second is parenthesized, it will be parsed as the second part of an expression using the corresponding binary operator. For example, `x -y` is ambiguous, but will be parsed as `x-y`; use `x (-y)` to cause the other interpretation.

References

[1] Altera. http://www.altera.com/support/software/nativelink/quartus2/glossary/def_mif.html.

[2] Intel. Intel hexadecimal object file, format specification revision a. *http://www.interlog.com/ speff/usefulinfo/Hexfrmt.pdf*, January 1988.

[3] Linux man page. srec - motorola s-record record and file format. *http://www.die.net/doc/linux/man/man5/srec.5.html*.

[4] Norman Ramsey and Mary Fernandez. The new jersey machine-code toolkit. *Proceedings of the 1995 USENIX Technical Conference*, pages 289–302, January 1995.

[5] IEEE Computer Society. IEEE P1364-2005/D2, standard for verilog hardware description language (draft). page 295, May 2003.

[6] Thomas G. Szymanski. Assembling code for machines with span-dependent instructions. *Commun. ACM*, 21(4):300–308, 1978.

[7] A. S. Tannenbaum. *Structured Computer Organization*. Prentice-Hall, 1976.

```
specifications : ( {specification } EOS )* MYEOF ;
specification :
 WORD ( punct | WORD | testexpr )* ASSIGN ( field )+
  | ASSIGN WORD ( ( ( WORD ) | ( NUM ) ) )*
  | ASSIGN NUM ( ( ( WORD ) | ( NUM ) ) )*
  | WORD SET expr | WORD EQU expr
  | AIKTextFormat ( EQU | SET ) WORD
  | AIKDataFormat ( EQU | SET ) WORD
  | AIKFormat ( EQU | SET ) WORD
  | AIKAsmExt ( EQU | SET ) WORD
  | AIKTextExt ( EQU | SET ) WORD
  | AIKDataExt ( EQU | SET ) WORD ;
testexpr : QUEST LPAREN expr RPAREN ;
field : expr COLON expr ;
instructions : ( { instruction } EOS )* MYEOF ;
instruction : ORG expr
  | IF expr | ELSE | END | TEXT | DATA
  | DW ( expr )+ | DS expr
  | WORD COLON { instruction }
  | WORD EQU expr | WORD SET expr
  | WORD ( punct | expr )* ;
expr : expr0 { QUEST expr0 COLON expr0 } ;
expr0 : expr1 ( OROR expr1 )* ;
expr1 : expr2 ( ANDAND expr2 )* ;
expr2 : expr3 ( OR expr3 )* ;
expr3 : expr4 ( XOR expr4 )* ;
expr4 : expr5 ( AND expr5 )* ;
expr5 : expr6 ( ( EQ expr6 ) | ( NE expr6 ) )* ;
expr6 : expr7 ( ( LT expr7 ) | ( GT expr7 )
         | ( LE expr7 ) | ( GE expr7 ) )* ;
expr7 : expr8 ( ( SHR expr8 ) | ( SHL expr8 ) )* ;
expr8 : expr9 ( ( ADD expr9 ) | ( SUB expr9 ) )* ;
expr9 : expra ( ( MUL expra ) | ( DIV expra )
          | ( MOD expra ) )* ;
expra : LPAREN expr RPAREN
  | ADD expra | SUB expra | NOT expra | LNOT expra
  | NUM | WORD | DOT ;
punct : "\$" | "\#" | "\[" | "\]" | "\," | "\@" ;
```

Figure 7: ANTLR Grammar For AIK