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

24th May 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. This document describes AIK version 20070512.

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 Tanenbaum'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.

```
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

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

Figure 2: AIK Specification Using Format Aliases

```
.const zero _ v0 v1 a0 a1 a2 a3
.const 8 t0 t1 t2 t3 t4 t5 t6 t7 24 t8 t9
.const 16 s0 s1 s2 s3 s4 s5 s6 s7
.const 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`

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`. 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 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 alias mechanism. This alias syntax not only allows a single instruction format specification to be parametrically used for many instructions, but also provides the ability to associate user names to built-in operations.

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 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 [5], 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 more straightforward use of `.alias` in AIK is to rename built-in operations. For example, simply saying "`.alias .origin ORG`" is how `ORG` can be used to mean the same thing as `.origin`. The `.alias` construct is actually general enough to able to be abused in a variety of

ways, because even operators can be aliased... but just because you can doesn't mean you should.

1.2 Constants

It is quite common in assembly languages that a sequence of symbolic names is used to represent a sequence of numeric constant values that are not instruction opcodes. The MIPS instruction set defines symbolic names for various general-purpose registers to suggest their recommended uses, but the symbolic names are really nothing more than predefined constants. For example, `$ra`, the "return address" register, is really `$31`. Rather than forcing a series of `.equate` operations to be specified, AIK's `.const` syntax allows arbitrarily many symbols to be given a sequence of values – essentially like `.alias` does for instructions, but without making the defined entities instructions. The values of the symbolic names for the MIPS registers can be specified as shown in Figure 3.

1.3 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.

Variable	Default	Meaning
`.this`		Refers to current instruction (opcode) value
`.passes`	`100`	Maximum number of analysis passes
`.lowfirst`	`0`	If non-zero, emit words in low-to-high order
`.version`	`20070512`	Date of current version release
`.requirecolon`	`1`	If non-zero, require `:` in label definitions

Table 1: Option Variables For Use In Specifications

1.4 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 `.equated` 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.

1.5 AIK Segment Definitions

Although Aik will create two segments, `.text` and `.data`, if the user does not specify others, the output segment names and their properties can be specified:

`.segment` ***name width depth baseaddress format*** Define an output segment to be entered by giving *name*; this *name* also is used as a suffix on the source file name to form the output file name for the segment. Within this segment, each address holds an object of *width* bits; this is treated as the "word" size of the segment. The *depth* is the number of addresses in the segment. The *baseaddress* sets the default origin of the segment. Finally, the format selects the format for the output associated with this segment. The current choices are:

`.I8HEX` Intel Hex format [3], with *width* no more than 8 bits.

`.SREC` Motorola Mikbug S records [4], with *width* no more than 8 bits.

`.MIF` Altera Memory Image File [1].

`.VMEM` Verilog MEMory image file [2].

1.6 AIK User-Defined Warnings & Errors

In specifying patterns, it often can be useful to output warnings or error messages when specific events are detected. For example, it would be nice to output a warning if a constant address used in a load word instruction was not properly aligned. For this purpose, the right side of a specification can contain any number of the following:

`.warn(`*expr*`, `*mesg*`)` If the value of *expr* is defined and non-zero, cause the warning *mesg* to be output. The message should be a string in double quote marks, `"like this."`

`.error(`*expr*`, `*mesg*`)` If the value of *expr* is defined and non-zero, cause the error *mesg* to be output. The message should be a string in double quote marks, `"like this."` Unlike `.warn`, `.error` will terminate the assembly.

2 Assembly Language Code

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, including the assignment operators. The only omission is that the pre-increment (`++`*x*) and pre-decrement (`--`*x*) operators have been disallowed to avoid ambiguity from the fact that, unlike C, AIK syntax allows two expressions to appear next to each other. The same functionality as `++`*x* can be unambiguously obtained by *x*`+=1`, and `--`*x* is *x*`-=1`; post-increment and post-decrement are both directly supported. There also are several additions to C expression syntax modeled after the extended expression syntax used in SWARC and BITC. The `?<` (minimum) and `?>` (maximum) binary operators have proven useful for many computations. Further, AIK provides `||=`, `&&=`, `?<=`, and `?>=` assignment operators with the obvious meanings.

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` Giving the name of any segment (such as the `.text` and `.data` segments automatically created if no others are specified) causes the segment named to be selected. Each segment has its own independent location counter. The first time a particular segment is entered, that segment's location counter is initialized to the segment's specified origin. Whenever a segment is entered, that segment's location counter becomes the active location counter.

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

`.word` ***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.

Operator	Level	Group	Meaning
x ? *y* : *z*	1		if (*x*!=0) return(*y*) else return(*z*)
x || *y*	2	>	if (*x*!=0) return(1) else return(*y*!=0)
x && *y*	3	>	if (*x*!=0) return(*y*!=0) else return(0)
x | *y*	4	>	return(bitwiseOR(*x*, *y*))
x ^ *y*	5	>	return(bitwiseXOR(*x*, *y*))
x & *y*	6	>	return(bitwiseAND(*x*, *y*))
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	>	if (*x*<*y*) return(*x*) else return(*y*)
x < *y*	8	>	if (*x*<*y*) return(1) else return(0)
x <= *y*	8	>	if (*x*<=*y*) return(1) else return(0)
x ?> *y*	8	>	if (*x*>*y*) return(*x*) else return(*y*)
x > *y*	8	>	if (*x*>*y*) return(1) else return(0)
x >= *y*	8	>	if (*x*>=*y*) return(1) else return(0)
x >> *y*	9	>	return(signedShiftRight(*x*, *y*))
x << *y*	9	>	return(signedShiftLeft(*x*, *y*))
x + *y*	10	>	return(add(*x*, *y*))
x - *y*	10	>	return(subtract(*x*, *y*))
x * *y*	11	>	return(multiply(*x*, *y*))
x / *y*	11	>	return(signedDivide(*x*, *y*))
x % *y*	11	>	return(signedModulus(*x*, *y*))
(*x*)	12		return(*x*)
x ++	12		t=*x*; *x*=*x*+1; return(t)
x --	12		t=*x*; *x*=*x*-1; return(t)
x op= *y*	12	<	*x* = *x op y*; return(*x*)
x = *y*	12	<	*x* = *y*; return(*x*)
+ *x*	12	<	return(*x*)
- *x*	12	<	return(subtract(0, *x*))
! *x*	12	<	if (*x*!=0) return(0) else return(1)
~ *x*	12	<	return(bitwiseNOT(*x*))

Table 2: AIK Expression Operators

`.space` ***expr*** Define a space of as many 0-valued words as the value of the expression *expr*.

lab **`.equate`** ***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 `.equate`, changes in the set value will not cause an additional pass. As a convenience, assignment operators, such as `=`, also may be used like `.set`.

`.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.

`.ifref` ***name*** Conditional assembly support; only generate code for the following assembly source if *name* has been referenced. The tricky part here is that name is not considered to be referenced by appearing in `.ifref` nor are apparent references to name counted if they occur within code that is conditionally disabled by `.if` or `.ifref`. In other words, `.ifref` makes it easy to conditionally assemble library code defining *name* if and only if *name* is referenced – and AIK even deals with rather nasty cases requiring multiple passes (i.e., those reference cycles that would require you to list a library multiple times on a conventional linker's command line). The happy result is that using `.ifref` ensures your program contains only the code and data it actually might use, which is really important for the type of systems AIK was designed to target; every library routine and data structure should be wrapped inside `.ifref`.

`.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` or `.ifref`.

`.align` ***expr*** Insert 0-valued words until the current location counter is a multiple of the value of expression *expr*. This is a very close cousin to `.space`, and could be implemented using an appropriate expression with `.space`, but this use is common enough to justify having a separate operation.

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 `.word`.

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.

```
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`

```
.Reg $rd,$rs,$rt := 0:6 rs:5 rt:5 rd:5 0:5 .this:6
.alias .Reg 32 add addu sub subu and or 42 slt sltu
.Imm $rt,$rs,imm := .this:6 rs:5 rt:5 imm:16
.alias .Imm 8 addi addiu slti sltiu andi ori
.Br $rs,$rt,lab := .this:6 rs:5 rt:5 ((lab-(.+4))/4):16
.alias .Br 4 beq bne
.LdSt $rt,imm[$rs] := .this:6 rs:5 rt:5 imm:16
.alias .LdSt 35 lw 43 sw
lui $rt,imm := 15:6 0:5 rt:5 imm:16
.Shift $rd,$rt,shamt := 0:6 0:5 rt:5 rd:5 shamt:5 .this:6
.alias .Shift sll 2 srl
.Jmp lab := .this:6 (lab/4):26
.alias .Jmp 2 j jal
jr $rs := 0:6 rs:5 0:5 0:5 0:5 8:6
.const { zero 2 v0 v1 a0 a1 a2 a3 t0 t1 t2 t3 t4 t5 t6 t7
         s0 s1 s2 s3 s4 s5 s6 s7 t8 t9 28 gp sp fp ra }
.segment .text 8 0x10000 0 .VMEM
.segment .data 8 0x10000 0 .MIF
```

Figure 7: AIK Specification Of The MIPS Subset Used In EE380

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. Optionally, a name can have `.` as its first character. Although the specification and assembly source sections of AIK input have the same namespace, it is strongly suggested that names starting with `.` only be defined in the specifications, thus providing a separate namespace by convention while allowing the separation to be overridden in the rare cases that is appropriate.

The basic MIPS instruction set is probably most familiar to most people, and thus serves as a good example. The subset used in the University of Kentucky's EE380 class is specified in Figure 7.

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 segments specified.

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.

This document was first written when AIK was very young – it had only been available to friendly users for a few weeks. Even in that short time, there were 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. This document will continue to be updated to reflect the current release version – which is indicated in the abstract.

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 8 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 caused by the fact that, unlike C, AIK allows two expressions to appear in sequence. If second one has a unary sign, 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] IEEE Computer Society. IEEE P1364-2005/D2, standard for verilog hardware description language (draft). page 295, May 2003.

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

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

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

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

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

```
specifications:({specification} EOS)* MYEOF;
specification:
 WORDINST (punct|WORDINST|testexpr)* IS (field {COMMA})+
|ALIAS BUILTIN . WildCard
|ALIAS WORDINST (((WORD)|(NUM)))*
|CONST ({(NUM)|(LPAREN expr RPAREN)} ((WORD)|(SUB)))+
|WORD SET expr|WORD ASSIGN expr|WORD EQU expr
|SEGMENT WORDSEGNAME expr expr expr FORMAT;
testexpr:QUEST expr;
field:expr {COLON expr}
|WARNERR LPAREN expr {COMMA} MESG RPAREN;
instructions:({instruction} EOS)* MYEOF;
instruction:((WORD COLON)|(WORD))* {generator}
|WORD EQU expr|WORD SET expr
|WORD ADDADD|WORD SUBSUB|WORD BUILTINASSIGN expr
|IF expr|IFREF WORD|ELSE|END
|ORG expr|SEGNAME;
generator:INST (punct|expr)*
|DW (expr)+|DS expr|ALIGN expr;
expr0:expr {QUEST expr COLON expr};
expr: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 ((MIN expr7)|(MAX 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 expr0 RPAREN
|ADD expra|SUB expra|NOT expra|LNOT expra|NUM|INST
|WORD ADDADD|WORD SUBSUB|WORD {BUILTINASSIGN expr}
|DOT|DEFINED WORD;
punct:"\$"|"\#"|"\["|"\]"|"\@"|COMMA;
```

Figure 8: ANTLR Grammar For AIK