Assemblers for firmware systems

Dr. Sergei A.Sidorov
Computational mathematics and cybernetics dept.,
Moscow State University

sidorov@niisi.msk.ru

One of the most important parts of firmware system is the machine level debugger. It includes command set, which are standard for the most of debuggers: set/clear breakpoint, see and modify values in memory or in processor registers, start and trace program, etc. Nearly all such debuggers have disassembler, but on-line assemblers occur rarely. Why? Obviously, we use on-line embedded assembler not so often as disassembler, and it is difficult and bulky task to create more or less full assembler, each time new for new architecture.

However, careful studying of this task shows that difficulties are overstated in many cases.

Let's divide this task into parts: translator «control» part and «executive» part. Control part includes input/output, errors processing, tables building, and other, and executive part means machine instructions translation itself. The first part is machine independent, the second part depends on processor architecture. At the same time, executive part includes a lot of universal procedures of syntactic parsing, table search, and implements, as a rule, the same set of assembler directives. These procedures may be named «service» part. As a result, we have three-level construction, where only middle level is fundamentally architecture-dependent, and others need small modification. In other words, once created assembler can be adopted by firmware designer to new architecture in a short time.

Author used this approach in assemblers creation for firmware systems, based on different architecture - 8080, Z80, 8051, PDP-11, MC68K, MIPS, and it justified hopes. As a base programming system for firmware designing and assemblers creation we used DSSP - Dialogue System for Structured Programming [1, 2, 3], but stated principles are independent of programming language.

Main assembler features

Creating firmware you mostly need to use cross-translator, which is not always available (especially it was actual before Internet age), too expensive or not exist yet. So it is useful for firmware designer to have his own cross-assembler, simple and satisfying his requirements. Two low levels of assembler, executive and service, are used in firmware embedded debugger. Let's look main features of such an assembler.

The first and constitutive in many reasons condition: the language of our assembler must be a subset of standard assembly language for given architecture. It is very important requirement. First, it gives opportunity to use this translator for different users (in addition to firmware author: hardware adjusters, operating system designers, first applied programmers) without studying once more language. Second, early or late the standard translator becomes available and usually have many advantages. If we wrote on standard language, we can start to use new translator very soon. Of course, it is not necessary to implement whole assembly language. What features to omit - it's up to you, usually they are: the most part of assembly directives, embedded macros or even whole class of machine instructions, for example, arithmetic co-processor.

There are some necessary assembly directives. They can be named differently, but essence is the same:

org	Set the origin value to instruction counter
equ	Define constant for translation time
include	Include text from another file
word, byte, ...	Define variables and memory spaces
align	Align instruction counter

Indispensable condition - full label processing, including forward and backward references. It relates with listing production. Usually it is not difficult to resolve all the references during the first translator pass, but in this case forward references are not appear in the listing. There are two ways: to produce code at the first pass and generate listing at the second optional pass; or make two-pass translator. Second way mostly easier and more compact, but slower if listing is not needed. Besides, on-line assembler can not be two-pass.

As an output the plane binary file is suitable enough. Code in this file must start from the first address set by org directive. Also it is applied to code generation in memory buffer.

It is a bad idea to be stingy in error diagnostics. Detail message and precise place pointing save much time during assembler usage. Two variants of diagnostics are preferred: short for on-line assembler - only error number, and full for cross-assembler. And what to do if error occurs: continue or abort translation? To reduce re-translating it is better to process whole text, because assembler works line by line with program text and lines usually independent one from another.

Control part

What does the control part of assembler do? The common scheme of assembler working is: input next program line, process this line, and so on to the end of program text, and output code.

Input and output streams. For cross-assembler input stream is the sequence of program lines which are read from files (with included ones). Line numbers must be separate in each file for precise error diagnostics. In embedded on-line assembler input stream is a line read from keyboard. In some cases it is useful to translate text from memory buffer (without nesting, of course). Loading program text to memory buffer is the individual task.

There are two output streams - code and listing. It is easier to write whole code at the translation end. Listing is generated line by line and one program line can produce several listing lines. Listing lines are output during translation.

Error diagnostics. This procedure is called by executive part when it finds an error. The single parameter passed is the error number, all other information is available in global variables: file name, line number, column and program line itself. Error processing procedure must print message in convenient form. In on-line assembler text message can be omitted (only error number) but error position must be pointed exactly.

Tables creation. At least two tables are needed in assembler: instruction mnemonics table and register names table. Often once more table is added - instruction modifiers (usually for co-processor). Table of labels and defined names is formed during translation. It is alluring idea to make common format for all the tables and to use universal search procedure, but in case if one-pass assembling label table structure can be complex, too bulky for mnemonics. There are many differences in information in these tables. In any case special procedures are needed for tables creation in such a «dynamic» languages as Forth and DSSP, or data structures if you use C, with search procedures. Fields access functions are architecture-dependent and belong to executive part.

References resolving. Two-pass translation supposes label table creation at the first pass and using this table at the second pass for code generation. One-pass translation assumes temporary collecting of information about forward reference: destination, source point and format. Using of undefined name causes creation of new table entry with «undefined» mark and list of places where this name is used. This list is used for resolving references when name is defined. Only reference format is architecture-dependent in this mechanism.

Executive part

In general assembly program line looks like:

<label> <instruction> <operands> <comment>

<label> is a name with «:» at the end. <instruction> is a name or composite name, i.e. two names linked by «.», for example move.b. <operands> is a comma separated «phrases», may be complex as alpha@(r2)+. <comment> is the rest of line, beginning with character «;», «#», «/» or something like this. Modern assembly languages often use comments /*...*/ which can occupy more than one line.

Line processing starts with variables initialization and line text preparing: transformation to small or capital characters and deletion of comments (BEGIN-LINE):

```
: ASM-LINE  BEGIN-LINE  LAB?  INSTRUCTION  END-LINE  ;
```

Further, if the first name has «:» at the end then it is a label, else it is instruction. Label is placed to name table (check redefinition!). One-pass translation assumes forward references resolving to this label, but it must be done at the end of line processing (END-LINE). True label value becomes available only at the end in case, for example, of label at org directive:

```
m:    org   0x100
```

Instruction name gives the key to all residuary processing. At the step of prior analysis of instruction set they all are divided on groups according to format. Usually big groups consist of arithmetic, logical and branch instructions. In some cases instruction name assumes addressing mode:

add - register,

addi - immediate.

Group includes instructions with all the same features except code.

Now we know number and types of operands we expect and we can start processing. Traditionally it is the most difficult point and the difficulty depends on processor architecture. Among known to author the most «involved» architecture was MC68K and the simplest was MIPS. In any case operand parsing consists of next steps: separate character or word, analyse it and define more precise operand type, and at the end form part of machine code. The most number of errors occurs during operand parsing.

In first versions it is admissible to limit expressions in operand with number, name and string. In future it is easy to change expression processing procedure to more powerful which allows arithmetic and logical operations, parenthesis.

Very important action in executive part creation is attentive analysis of instruction set, groups and instruction table forming. Group must include only instructions with exactly the same format. For example, MIPS instructions add and sllv have very similar format

add rd,rs,rt and sllv rd,rt,rs

but they belong to different groups. At the same time processing procedures consist of identical parts (in Forth or DSSP):

```
: ADD  SETRD KNXT, SETRS KNXT, SETRT ,KOP ;
: SLLV SETRD KNXT, SETRT KNXT, SETRS ,KOP ;
```

SETRx gets from input line the next word, checks it for register name and puts it number to one of three fields in instruction code. KNXT, kills comma between operands, and ,KOP writes ready instruction code.

Register table is easy. It keeps register names, numbers and, may be, co-processor number. It is useful to allow redefinition of register name, for example, with directive

renr <old name> <new name>

Service part

Syntactical analysis of assembly language is easy enough. All the information we can get from input line. There is the pointer to first unprocessed character. Syntactical analysis is based on several procedures which are enough for parser:

SKIP-SP	(--)	Skip all spaces
NXTS	(-- c)	Push next character and move pointer
?NXTS	(-- flag)	Push next character without pointer movement (peep)
KNXTS	(--)	Skip next character
KNXT,	(--)	Check comma and skip it
@NAME	(-- a l)	Get next name
@EXPR	(-- a l)	Get next expression
EOL?	(-- flag)	Check end of line
FIND-INSTR	(-- group)	Find instruction in table by name
FIND-REG	(-- num)	Find register in table by name

Next level procedures do more complex actions:

?LAB	(--)	Read the first name, check for label and put it into table
EVAL-EXPR	(a l -- p)	Evaluate expression. Expression value is put into variable(s) and p means: 1 - new name, 2 - known name (number or name value), 3 - text string, 4 - used but undefined name, . . .
DO-LAB	(--)	Process label at the end of line processing: assign true value and resolve references to this label.
ORG EQU SPACE	(--)	Do assuming actions. The same directives can have different names in assemblers.
BYTE WORD LONG	(--)	Define data of given size. Operands are the comma separated list of expressions.

. . .

It is obvious that service procedures do the main part of bulky and tedious work. They are independent of computer architecture, but sometimes small tuning is needed (add or remove data format, use quotation marks or apostrophes for test strings, etc).

References

1. Sidorov S.A., Shumakov M.N. DSSP and Forth. Compare analysis. // 12th EuroFORTH conference on the FORTH programming language and FORTH processors. St.Peterburg, Russia. 1996 (11 pp.).

2. Sidorov S.A. Data in DSSP - prefix access in postfix language. // EuroFORTH'97. Conference proceedings.- Oxford, England, 1997 (7 pp.).

3. Sidorov S.A. Top-down Thinking and Top-down Writing in DSSP. // EuroFORTH'98: 14th Euroforth Conference, Schloss Dagstuhl, Germany, 1998 (5 pp.).