Computer Science 237
Computer Organization
Williams College
Fall 2006

The WC34000 Assembler

Just as the WC34000 computer supports a subset of the features of the 68000, the assembler provided for the 34000 supports a subset of the language accepted by the 68000 assembler on the Palm. This document describes the language accepted by the 34000 assembler and explains how to use the assembler.

Using the Assembler

To assemble a program, simply type

```
wc34asm file-name
```

where *`file-name`* is the name of a file containing assembly language statements of the form described below. The assembler will display error messages on the error output. If no errors are encountered the assembler will place the translated program in a file named '`tmem`' in the current directory. The `-l` option can be used to request that the assembler produce a listing of the program on its standard output. Thus,

```
wc34asm -l prog.s > prog.l
```

will assemble `prog.s` leaving a listing of the program in `prog.l` and the translated program in `tmem`. The listing will show the address of the operation field of each machine instruction in the program. This information can be very useful when debugging a program.

The `tmem` File

The `tmem` file contains a binary image to be loaded into the 34000's memory. When the file is loaded, memory word 0 will contain the address of the first instruction or the program to be executed and memory word 1 will contain the base address of the global variable area. The machine's hardware uses the contents of word 0 to determine where to begin execution. The executed program is responsible for loading the contents of word 1 into register A5.

Program Format

A 34000 assembly language program is just a file containing a sequence of *instructions* and *directives*. All characters on a line including and following the occurrence of a semicolon are treated as a comment to be ignored by the assembler.

The general form of an instruction is:

label op-code operands

The label field is optional.

The 34000 assembler recognizes any sequence of alphabetic and numeric characters that starts with an alphabetic character as a valid label. No special characters can be used in labels (i.e. '.', '$' and '_' are not allowed). The case of characters used in labels is significant.

The positioning of instruction components on a line is not restricted. Blanks between identifiers, constants and delimiters are ignored. In particular, labels need not be placed in the first column or followed by a colon.[1] To allow it to distinguish labels from operation codes, the assembler treats all operation codes as reserved symbols. That is, they can not be used as labels. Case is not significant in operation codes.

Operand Format

The 34000 assembler differentiates syntactically between memory operands and register operands. In certain contexts, only a memory operand can be used. In particular:

- The operand of a jump or branch instruction must be a memory operand.
- The operand of a push effective address (`pea`) instruction must be a memory operand.
- The first operand of a load effective address instruction (`lea`) must be a memory operand.

In addition, the first operand of a link instruction must be an address register name.

Register Operands

Register operands are specified using the name of the register desired. The data registers are named D0, D1, ... D7. The address registers are named A0, A1, ... A7. SP is another name for A7. The case of alphabetic characters used in register names is not significant.

Memory Operands

The following forms are recognized as valid memory operand specifications:

[1] However, labels may be followed by colons.

`(An)`	- Address register indirect
`--(An)`	- Address register indirect with predecrement
`(An)+`	- Address register indirect with postincrement
`#constant`	- Immediate data
`label` `constant`	- Absolute addressing
`label`	- Program counter indirect with displacement

`label`
`label(An)`
`constant(An)`
	- Address register indirect with displacement

As indicated above, the use of an operand specification of the form

```
label
```

can cause the assembler to generate an instruction using one of three different addressing modes depending on the type of label used. If the label was defined by an `EQU` directive, the assembler uses absolute addressing. If the label was defined in a `GLOBAL` directive, the assembler uses address register relative addressing (relative to A5). Finally, if the label was defined by its appearance in the label field of an instruction or an `ASCII` directive, an effective address using program counter relative addressing will be generated. If an identifier defined as a global or as a statement label is used in an operand specification of the form:

```
label(An)
```

the displacement generated will be the same as if the identifier had been used alone.

Constants in Operands

In those operand formats that require a constant, the assembler will accept either an integer or character constant. (String constants are also supported by the assembler, but they are only valid in `ASCII` directives as discussed below). Any sequence of numeric characters optionally preceded by a '`+`' or '`-`' that is not part of an identifier is recognized as an integer constant. Any single printable character (except `'` or `\`) surrounded by single quotes (`'`) is treated as a character constant. In addition, certain escape sequences are recognized for character constants. The form of these escape sequences and the characters they represent are shown below:

Sequence	Value
'\''	Single quote
'\n'	Newline
'\t'	Horizontal Tab
'\\'	Backslash

Directives

Directives instruct the assembler in various ways, but do not lead to the production of machine code statements. The 34000 assembler recognizes several directive: `EQU`, `GLOBAL`, `ASCII`, `SOURCE` and `STAB`.

The `EQU` Directive

An `EQU` directive takes the form:

label equ *constant*

The `EQU` directive associates the label given with the constant specified. Note that in the assembler, `EQU`'s cannot be used to associate symbolic names with registers.

The `GLOBAL` directive

The `GLOBAL` directive replaces the `DS` directive provided by most 68000 assemblers as the means to allocate storage for global variables. A `GLOBAL` directive takes the form:

`global` *declaration-list*

The declaration list is a list of declarations separated by commas. A declaration is either simply an identifier or an identifier followed by a parenthesized integer constant. Specifying a simple identifier as a declaration in a `GLOBAL` directive asks the assembler to associate the identifier used with a single word of memory. Specifying an identifier followed by a constant causes the assembler to reserve a number of words equal to the constant's value and associate the identifier with the first word reserved.

For example, the global directive

```
global a, b, c(10), d
```

Causes the allocation of a word for each of the names '`a`', '`b`' and '`d`' and the allocation of an array of 10 words for '`c`'.

More than one global declaration may appear in a program.

The ASCII Directive

The ASCII directive is used to allocate and initialize string constants in memory. The form of an ASCII directive is

label ASCII *string constant*

This directive tells the assembler to store the characters appearing in the string constant in the memory locations immediately following those used to store the code generated by the previous machine instruction (or the character values stored as a result of a previous ASCII directive). Each character in the string is stored in one word of memory. A word containing the value 0 is placed after the last character of the string. The label is associated with the word containing the first character of the string.

The string constants used in ASCII directives are formed by placing double quotes (") before and after any string of printable characters (except for double quotes and backslashes). Within a string, any of the escape sequences allowed in character constants may be used. In addition, the sequence \" may be used to include a double quote in a string.

The SOURCE and STAB Directives

The SOURCE directive and the STAB directive are used to provide information to the assembler about a high-level language source file from which an assembly language file was created. In particular, the C-- compiler for the WC34000 includes SOURCE and STAB directives including enough information to allow the WC34000 debugger to do source-level tracing and to display variables defined in the source program.

The SOURCE directive is very simple. It takes the form

SOURCE *line number*

Such a directive informs the assembler that all following assembly language instructions up to the next SOURCE directive were generated as the translation of the specified line number in the original source file.

The general form of an STAB directive is:

STAB *arguments*

The arguments to an STAB are separated by commas.

The STAB directive comes in many varieties. The first argument identifies the type of each STAB directive. Each of the types is discussed below.

STAB **Source File Specification**

If the first argument to an STAB directive is the word "FILE", the STAB is used to identify the name of the source file from which this assembly language file was generated. The file name appears as a quoted string in the second argument position.

STAB **Global Variable Descriptions**

STAB directives are used to provide debugger symbol table information about global variables defined in the source program. The word "GLOBVAR" is used as the first argument of such a directive. The second argument must be a quoted string specifying the variables name. The third and fourth arguments are specified using a keyword format. The third argument must be of the form OFFSET=n where n is an integer constant specifying the offset to the storage used to hold the variable's value from the base of the global variable area. The fourth argument must be of the form TYPE="type-string" where "type-string" specifies the variables type.

The format used for type-strings is designed more for easy digestion by the debugger than for ease of reading. The single letters "I" and "C" are used to specify the types integer and character. A "*" followed by a type string is used to identify a pointer to a value of the type described by the type string. Thus, the type string "*I" would be used for a variable defined as a pointer to an integer. Array types are indicated by two integers in square brackets followed by a type string. The first integer indicates the number of elements in the array. The remainder of the type string indicates the type of the array's elements. The second integer indicates the size of each value of the element type. Thus, "[10,1]*I" would describe an array of 10 integer pointers. Finally, structure and union types are specified by including an integer in curly braces (i.e. "{4}"). In this case, the integer is interpreted as the number associated with the structure/union type in the STAB-STRUCT directive that describes the type to the debugger.

STAB - **Function Specifications**

If the first argument to an STAB directive is the word "FUNCTION", the directive indicates that the code following the directive up until the next STAB-FUNCTION directive was produce to implement the function whose name appears as the second argument. The name must be specificed as a quoted string.

STAB - **Local Variable Descriptions**

Local variables are described using STAB directives that are very similar to those used for global variables. The first argument of such a directive is the word "LOCVAR" rather than "GLOBVAR" and the offset is specified relative to the function activation record with which the variable is associated. Otherwise the directive is identical to a GLOBVAR directive. The second argument specifies the variable name and the fourth the variable's type.

The assembler associates each STAB-LOCVAR directive it encounters with the function described by the most recently processed STAB-FUNCTION directive. It is an error to include an STAB-LOCVAR directive before the first STAB-FUNCTION directive.

STAB - Structure and Union specifications

Each structure or union type used in the source program is described by a STAB-STRUCT directive followed by a sequence of STAB-COMPONENT directives. The STRUCT directive consists of the word "STRUCT" as first argument followed by an integer constant as the second argument. Each structure/union type must have a unique integer constant associated with it. This number is used to refer to the type in the type specifications found in LOCVAR, GLOBVAR and COMPONENT directives.

Components are described using an STAB directive with the word "COMPONENT" as its first argument. The other arguments to such a directive are like those found in LOCVAR and GLOBVAR directives. The second argument is the component's name (in quotes), the third is the offset to the component (relative to the beginning of the structure) and finally the fourth is the type of the component. Each component is associated with the structure specified by the last STAB-STRUCT directive processed.