RICHMOND School of Arts & Sciences

University of Richmond
UR Scholarship Repository

Math and Computer Science Technical Report Series

Math and Computer Science

8-10-1994

A Macro Extension for the Woody Assembly Language

Lewis Barnett III
University of Richmond, lbarnett@richmond.edu

Follow this and additional works at: http://scholarship.richmond.edu/mathcs-reports

 Part of the Programming Languages and Compilers Commons

Recommended Citation

Lewis Barnett. *A Macro Extension for the Woody Assembly Language*. Technical paper (TR-94-06). *Math and Computer Science Technical Report Series*. Richmond, Virginia: Department of Mathematics and Computer Science, University of Richmond, August, 1994.

This Technical Report is brought to you for free and open access by the Math and Computer Science at UR Scholarship Repository. It has been accepted for inclusion in Math and Computer Science Technical Report Series by an authorized administrator of UR Scholarship Repository. For more information, please contact scholarshiprepository@richmond.edu.

A Macro Extension for the Woody Assembly Language

Lewis Barnett
Department of Mathematics and Computer Science
University of Richmond
Richmond, Virginia 23173
email: barnett@cs.urich.edu

August 10, 1994

TR-94-06

A Macro Extension for the Woody Assembly Language

Lewis Barnett

August 10, 1994

1 Introduction

We discuss an extension to the Woody Assembly Language [Cha94] which allows new instructions to be defined. The mechanism is similar to the C language's #define macros, allowing a name to be supplied for a piece of code which will be expanded in line. Provisions are made for writing new non-destructive branching instructions as well as instructions which are simply new names for commonly used bits of code.

2 Syntax

Each new definition should be enclosed in a `DEFINE/END` pair. The syntax of the `DEFINE` line is

`DEFINE` *name* [*label*]

where *name* is the name by which the definition can be referred to in a program, and *label* is an optional label which refers to a memory location. The label is a parameter for the definition. When a reference to the named instruction is expanded in line, the label in the definition will be replaced with the label supplied in the reference. The syntax of the `END` line is

`END` *name*

where *name* must match the name given in the corresponding `DEFINE` line.

Any valid Woody instruction can appear between the `DEFINE` and `END` lines. Reference to previously defined macros are also allowed in definitions. In addition to straight assembly language code (which can make reference to the *label* parameter and any label present in the program where the macro is referenced), instructions can also refer to a temporary memory location which will be automatically generated by the expansion process. One such temporary variable is allowed per definition, and is referred to as "$TEMP". If the last line of the definition is "@", the expansion of the definition in

line will also cause the code at location *label* to be changed to restore the contents of the Data Register from the automatically generated temporary storage location created by a previous reference to "**$TEMP**" in the definition. This feature allows new branch instructions which preserve the current state of the register to be written.

Definitions may also make reference to labels **$NEG**, **$ZERO** and **$ONE**. Memory locations with these names and the values -1, 0, and 1, respectively, will be automatically generated during the expansion process.

The **#include** directive is supported to indicate that a file consisting of macro definitions is to be loaded. Its syntax is

#include *filename*

where *filename* is the name of the file containing the definitions in the syntax of the computer file system Woody is running on.

Examples:

```
DEFINE ReadTo target
   CopyTo $TEMP
   Read
   CopyTo target
   CopyFrom $TEMP
END ReadTo

DEFINE GoTo target
   CopyTo $TEMP
   CopyFrom $NEG
   IfNegGoto target
   @
END GoTo
```

3 Usage

Any macro definitions referred to in a program must either be previously loaded before the program is executed or must precede the program in the file where it is stored. The first instruction which appears outside a **DEFINE/END** pair is assumed to be the first instruction of the program. Macro definitions are allowed to reference other macro definitions as long as those definitions precede the reference in the file. Self-reference is not allowed.

4 Implementation

The macro language is implemented in the HyperCard version of Woody for the Apple Macintosh in versions 2.0 and later. During a session (in which many different programs may be executed) the program builds up a "library" of macro definitions which may be referred to in the programs that are executed. Definitions may be incorporated in the library in one of three ways. First, definitions may appear in the same file as a program to be executed. Second, definitions may be placed in a separate file which is then included in a program file using the `#include` directive as described in Section 2. Third, definitions in a separate file may be loaded directly using a menu item. The first and second methods are preferred, since they make explicit the dependence of a program on the macro definitions it needs to execute.

Macro expansion takes place in two stages. First, the definitions are parsed and stored in a macro library. Second, macro references in the program itself are expanding according to the definitions in the macro library. All macros in the library are available to any program that is subsequently executed. If this is not the desired behavior, a menu item is provided to empty the macro library.

The first stage requires that all macro definitions be collected into a macro library. The form of the definitions which appear in programs or macro files is shown in Section 2. Each definition is stored in the macro library as a single line in a HyperTalk container with the form

name|*instr-1*|*instr-2*|... |*instr-n*[|@]

Where *name* is the name used to refer to the definition, *instr-i* is the i^{th} instruction in the definition, and @ is the symbol indicating that the Data Register should be restored at the target address for the reference to the macro being expanded. Any occurrence of the target address in the definition is changed to "!". References to `$TEMP` remain unchanged in the library version of the definition. At this stage, any definitions in `#include` files referenced by the program or macro file are also added to the library. Adding definitions to the library can be initiated either by requesting that references in a program file be expanded or by requesting that a macro file be loaded. Recall that macro definitions can themselves reference other macros. These references are left as is in the library version of the definition, since their expansion must be specific to the context in which the macro containing them is referenced.

The second stage is expanding macro references in the program itself. This in itself requires two passes through the source, the first to expand the references to the corresponding definition where the reference is made, and the second to patch up the source at target addresses where definitions required that the Data Register be restored. Recall that this may be necessary to implement new branch instructions which preserve the value of the Data Register, for example. In the first pass, each line of the program is examined to determine whether it contains a regular instruction or a macro reference. For those that do contain macro references, the reference is expanded from the library definition as follows: The reference is removed from the program and the address part is saved. Each instruction from the library definition of the macro is inserted into the

program. For each of these instructions, the operation is tested to determine if it is a regular operation or a nested macro reference; in the latter case, the reference is expanded at this point. If the address part of the instruction is a reference to the target address from the macro reference (indicated by a "!"), the saved target address is substituted. If the address part is a reference to `$TEMP`, an appropriate temporary variable name is generated and substituted. Temporary variable generation is straightforward: a global counter is maintained, which is appended to `$TEMP` to form the label. At the end of each expansion, the value of the counter is incremented, so that the next expansion which requires a temporary variable will use a different name. If the final instruction in the library definition is "@", then the saved target address and the generated temporary variable are saved in a list of locations where the code must be modified to restore the Data Register.

Since the location referenced by a macro which requires restoration of the Data Register may be earlier in the program than the location where the reference occurred, a second pass through the program is required to complete the expansion of macros. During this pass, the table of target addresses and temporary variables collected during the first pass is used to update the code. For each line of the program, the label is checked to see if it occurs in the table. If it does, code to restore the data register from the temporary variable specified in the table is added. Finally, the temporary variable index is checked to see if any temporary variables were used and thus need to be added to the end of the program.

5 Limitations

The current implementation allows only a single use of the `$TEMP` temporary label per definition, which is understood to be used for restoration of the contents of the data register if the definition alters it. This precludes the use of labeled memory locations within a macro definition. For example, there is no way to create a label for an instruction in a definition (to allow a loop within a definition, for example) which will be unique for each expansion of the macro in a program.

No provision is made for automatically and uniquely labeling the instruction following a macro reference. This facility would be useful in defining additional branching instructions.

References

[Cha94] Arthur Charlesworth. Nary One Bit O' Magic: How Computers Work. (produced by the University of Richmond print shop), 1994.