**Writing and Using Macros**

**.macro, .end\_macro,.eqv and .include directives are new in MARS 4.3**

**Introduction to macros**

Patterson and Hennessy define a **macro** as *a pattern-matching and replacement facility that provides a simple mechanism to name a frequently used sequence of instructions* [1]. This permits the programmer to specify the instruction sequence by invoking the macro. This requires only one line of code for each use instead of repeatedly typing in the instruction sequence each time. It follows the axiom "define once, use many times," which not only reduces the chance for error but also facilitates program maintenance.

Macros are like procedures (subroutines) in this sense but operate differently than procedures. Procedures in MIPS assembly language follow particular protocols for procedure definition, call and return. Macros operate by substituting the macro body for each use at the time of assembly. This substitution is called *macro expansion.*. They do not require the protocols and execution overhead of procedures.

As a simple example, you may want to terminate your program from a number of locations. If you are running from the MARS IDE, you will use system call 10, exit. The instruction sequence is pretty easy

li $v0,10

syscall

but still tedious. You can define a macro, let's call it done, to represent this sequence

.macro done

li $v0,10

syscall

.end\_macro

then invoke it whenever you wish with the statement

done

At assembly time, the assembler will replace each occurrence of the statement done with the two-statement sequence

li $v0,10

syscall

This is the macro expansion. The runtime simulator is unaware of macros or macro expansion.

If running MARS from the command line, perhaps you want to return a termination value. This can be done with syscall 17, exit2, which takes the termination value as an argument. An equivalent macro, let's call it terminate would be

.macro terminate (%termination\_value)

li $a0, %termination\_value

li $v0, 17

syscall

.end\_macro

This macro defines a *formal parameter* to represent the termination value. You would invoke it with the statement

terminate (1)

to terminate with value 1. Upon assembly, the statement terminate (1) would be replaced by the three-statement sequence

li $a0, 1

li $v0, 17

syscall

The *argument value*, 1, is substituted wherever the formal parameter %termination\_value appears in the macro body. This is a textual substitution. Note that in this example the argument value must be an integer, not a register name or a label, because the parameter is used as the second operand in the Load Immediate operation.

In MARS, a macro is similar to an extended (pseudo) instruction. They are distinguished in that the expansion of extended instructions is supported by an internally-defined specification language and mechanism which can manipulate argument values. The macro facility can only substitute argument values as given, and it uses a separate mechanism from extended instructions.

Additional examples and details follow.

**How to define macros**

The first line begins with a .macro directive followed by an optional list of formal parameters. Placing commas between parameters and parentheses around the list is optional.

Each formal parameter is an identifier that begins with a % character. For compatibility with the SPIM preprocessor APP, it may alternatively begin with $.

The lines that follow define the body of the macro. Use the formal parameters as appropriate. The body may contain data segments as well as text segments.

The macro definition finishes with a .end\_macro directive.

See the Notes below for additional information.

**How to use macros**

To invoke a macro, form a statement consisting of the macro name and then one token for each argument to be substituted for its corresponding formal parameter by the assembler. The argument list may optionally be surrounded by parentheses. Arguments may be separated either by spaces or commas.

Macro expansion is a pre-processing task for assemblers.

**Notes**

* A macro definition must appear before its use. No forward references.
* All macro definitions are local in each file and they cannot be global.
* Nested macro definitions are not supported. No .macro directive should appear inside body of a macro definition.
* A macro definition can contain a call to a previously-defined macro. Only backward references are allowed.
* Labels placed in the body of a macro definition will not have same name after macro expansion. During expansion, their name will be followed by "\_M#" where # will be a unique number for each macro expansion.
* Two macros with the same name but different number of parameters are considered different and both can be used.
* A macro defined with the same name and same number of parameters as another macro defined before it will be ignored.
* Each argument in a macro call can only be a single language element (token). For instance "4($t0)" cannot be an argument.
* Macros are a part of the assembler, not the ISA. So the syntax might be different with other assemblers. For compatibility with the SPIM simulator, *SPIM-style macros are also supported in MARS*. SPIM-style macros are same as MARS but formal parameters begin with "$" instead of "%".

**Examples**

* Printing an integer (argument may be either an immediate value or register name):
* .macro print\_int (%x)
* li $v0, 1
* add $a0, $zero, %x
* syscall
* .end\_macro
* print\_int ($s0)
* print\_int (10)
* Printing a string (macro will first assign a label to its parameter in data segment then print it):
* .macro print\_str (%str)
* .data
* myLabel: .asciiz %str
* .text
* li $v0, 4
* la $a0, myLabel
* syscall
* .end\_macro
* print\_str ("test1") #"test1" will be labeled with name "myLabel\_M0"
* print\_str ("test2") #"test2" will be labeled with name "myLabel\_M1"
* Implementing a simple for-loop:
* # generic looping mechanism
* .macro for (%regIterator, %from, %to, %bodyMacroName)
* add %regIterator, $zero, %from
* Loop:
* %bodyMacroName ()
* add %regIterator, %regIterator, 1
* ble %regIterator, %to, Loop
* .end\_macro
* #print an integer
* .macro body()
* print\_int $t0
* print\_str "\n"
* .end\_macro
* #printing 1 to 10:
* for ($t0, 1, 10, body)

The for macro has 4 parameters. %regIterator should be the name of a register which iterates from %from to %to and in each iteration %bodyMacroName will be expanded and run. Arguments for %from and %to can be either a register name or an immediate value, and %bodyMacroName should be name of a macro that has no parameters.

**Macro source line numbers**

For purpose of error messaging and Text Segment display, MARS attempts to display line numbers for both the definition and use of the pertinent macro statement. If an error message shows the line number in the form "X->Y" (e.g. "20->4"), then X is the line number in the expansion (use) where the error was detected and Y is the line number in the macro definition. In the Text Segment display of source code, the macro definition line number will be displayed within brackets, e.g. "<4>", at the point of expansion. Line numbers should correspond to the numbers you would see in the text editor.

**The .eqv directive**

The .eqv directive (short for "equivalence") is also new in MARS 4.3. It is similar to #define in C or C++. It is used to substitute an arbitrary string for an identifier. It is useful but much less powerful than macros. It was developed independently of the macro facility.

Using .eqv, you can specify simple substitutions that provide "define once, use many times" capability at assembly pre-processing time. For example, once you define

.eqv LIMIT 20

.eqv CTR $t2

.eqv CLEAR\_CTR add CTR, $zero, 0

then you can refer to them in subsequent code:

li $v0,1

CLEAR\_CTR

loop: move $a0, CTR

syscall

add CTR, CTR, 1

blt CTR, LIMIT, loop

CLEAR\_CTR

During assembly pre-processing, the .eqv substitutions will be applied. The resulting code is

li $v0,1

add $t2, $zero, 0

loop: move $a0, $t2

syscall

add $t2, $t2, 1

blt $t2, 20, loop

add $t2, $zero, 0

which when run will display the values 0 through 19 on one line with no intervening spaces.

Note that the substitution string is not limited to a single token. Like .macro, .eqv is local to the file in which it is defined, and must be defined prior to use. Macro bodies can contain references to .eqv directives.

**The .include directive**

The .include directive is also new in MARS 4.3. It has one operand, a quoted filename. When the directive is carried out, the contents of the specified file are substituted for the directive. This occurs during assembly preprocessing. It is like #include in C or C++.

.include is designed to make macro and equivalence (.eqv directive) use more convenient. Both macro definitions and equivalence definitions are *local*, which means they can be used only in the same file where defined. Without .include, you would have to repeat their definitions in every file where you want to use them. Besides being tedious, this is poor programming practice; remember "define once, use many times." Now you can define macros and equivalences in a separate file, then include it in any file where you want to use them.

The .include preprocessor will detect and flag any circular includes (file that includes itself, directly or indirectly).

The use of .include presents some challenges for error messaging and for source code numbering in the Text Segment display. If a file being included has any assembly errors, the filename and line number in the error message should refer to the file being included, not the file it was substituted into. Similarly, the line number given in the Text Segment source code display refers to the line in the file being included. Thus the displayed line numbers do not monotonically increase - this is also the case when using the "assemble all" setting. Line numbers should correspond to the numbers you would see in the text editor.

As a simple example, you could define the done macro (and others) in a separate file then include it wherever you need it. Suppose "macros.asm" contains the following:

.macro done

li $v0,10

syscall

.end\_macro

You could then include it in a different source file something like this:

.include "macros.asm"

.data

value: .word 13

.text

li $v0, 1

lw $a0, value

syscall

done

During assembly preprocessing, this would be expanded to

.macro done

li $v0,10

syscall

.end\_macro

.data

value: .word 13

.text

li $v0, 1

lw $a0, value

syscall

done

The assembler will then perform the appropriate macro expansion.

**Acknowledgements**

The MARS macro facility was developed in 2012 by Mohammad Hossein Sekhavat, sekhavat17@gmail.com, while an engineering student at Sharif University in Tehran. MARS creators Pete and Ken are incredibly grateful for his contribution! Pete developed .eqv and .include at about the same time.

**References**

[1] *Computer Organization and Design: The Hardware/Software Interface, Fourth Edition,* Patterson and Hennessy, Morgan Kauffman Publishers, 2009.