**MAL Code**

Time to write some MIPS assembly language code. Here are examples, to guide this effort.

**Conditional Execution**

Conditional execution is the equivalent of a high level language if statement. Sometimes an instruction (or a set of instructions) should be executed, and sometimes it (they) should not.

**general format of Pascal if-then-else**

if (condition) then

statement

else

statement;

**general format of C or JAVA if-then-else**

if (condition)

statement;

else

statement;

**If statement example:**

# MAL code fragment for the C or Java code:

#

# if (count < 0)

# count = count + 1;

#

lw $8, count

bltz $8, ifstuff

b endif

ifstuff: add $8, $8, 1

endif: sw $8, count

# next program instruction goes here

OR,

lw $8, count

bgez $8, endif

add $8, $8, 1

endif: sw $8, count

# next program instruction goes here

And, if variable count is already in register $8, and intended to be re-used, then count does not need to be loaded and stored:

bgez $8, endif

add $8, $8, 1

endif: # next program instruction goes here

**Examples of compound conditional:**

# MAL code fragment for the C or Java code:

#

# if ( (x < y) || (w == z) ) {

# a = a + 1;

# }

#

lw $8, x

lw $9, y

lw $10, w

lw $11, z

blt $8, $9, increment # no need to check second

bne $10, $11, no\_increment # condition if first is True

increment: lw $12, a

add $12, $12, 1

sw $12, a

no\_increment: # next program instruction goes here

# MAL code fragment for the C or Java code:

#

# if ( (x < y) && (w == z) ) {

# a = a + 1;

# }

#

lw $8, x

lw $9, y

lw $10, w

lw $11, z

bge $8, $9, no\_increment # must check second

bne $10, $11, no\_increment # condition if first is True

lw $12, a

add $12, $12, 1

sw $12, a

no\_increment: # next program instruction goes here

**Loops**

Structured loops can be built out of IF's and GOTO's.

**while loop example:**

# MAL code fragment for the C or Java code:

#

# while (count > 0) {

# a = a % count;

# count--;

# }

#

# Assume that $8 has variable count

# $9 has variable a

#

while: blez $8, endwhile

rem $9, $9, count

sub $8, $8, 1

b while

endwhile: # next program instruction goes here

**do while loop example:**

(NOTE: This example shows an implementation of nonsense code.)

# MAL code fragment for the C or Java code:

#

# /\* do statement while expression is TRUE \*/

# /\* when expression is FALSE, exit loop \*/

# do {

# if (aa < bb)

# aa++;

# if (aa > bb)

# aa--;

# } while( aa != bb);

#

# Assume that $8 has variable aa

# $9 has variable bb

#

repeat: bge $8, $9, secondif

add $8, $8, 1

secondif: ble $8, $9, until

sub $8, $8, 1

until: bne $8, $9, repeat

**while loop example:**

# MAL code fragment for the C or Java code:

#

# while ( (count < limit) && (c==d) ) {

# /\* loop's code goes here \*/

# {

#

# Assume that $8 has variable count

# $9 has variable limit

# $10 has variable c

# $11 has variable d

#

while: bge $8, $9, endwhile

bne $10, $11, endwhile

# loop's code goes here

b while

endwhile:

**for loop example:**

# MAL code fragment for the C or Java code:

#

# for ( i = 3; i <= 100; i++) {

# a = a + i;

# }

#

# Assume that $8 is the loop induction variable i

# $9 has variable a

# $10 is the constant value 100

li $10, 100 # set constant

li $8, 3 # initialize loop induction variable

for: bgt $8, $10, endfor

add $9, $9, $8

add $8, $8, 1 # increment loop induction variable

b for

endfor:

Putting this all together, we can write programs. Here is a sample MAL program.

# this simple MAL program reads in 2 characters, figures

# out which one is alphabetically first, and prints it out.

# register assignments

# $8 -- the first character typed by the user

# $9 -- the second character typed by the user

# $10 -- temporary

# $11 -- holds the value of the larger character

# $13 -- the address of the newline character constant

# $14 -- newline character (a constant)

.data

newline: .byte '\n'

.text

\_\_start: getc $8 # get 2 characters

getc $9

la $13, newline # print out newline

lb $14, ($13)

putc $14

sub $10, $9, $8 # figure out which is larger

bgez $10, secondlarger

add $11, $8, $0

b printresult

secondlarger: add $11, $9, $0

printresult: putc $11

end:

done

There are several things to notice about this program.

* Variables that are in memory are declared in the .data section of the program. This section typically appears first within the source code file.
* The code itself is in the .text section.
* The first instruction to be executed within the code is identified (in our simulator) by the special label \_\_start. This special label has 2 underscore characters in it.
* Comments do *not* span lines.
* When writing in assembly language, variables often reside in registers. This makes it difficult to figure out what the code is to do, without knowing which variables are in which registers. Therefore, by convention, always include comments that identify which variable is in which register.

**A program example that prints the alphabet:**

# MAL program to print out the alphabet

.data

str1: .asciiz "The alphabet:\n"

# register assignments

# $8 -- the ASCII character code to be printed

# $9 -- the ASCII code for 'z', the ending character

.text

\_\_start: la $10, str1

puts $10

add $8, $0, 97 # $8 gets ASCII code for 'a'

# could be li $8, 97

add $9, $0, 122 # $9 gets ASCII code for 'z'

# could be li $9, 122

while: bgt $8, $9, all\_done

putc $8

add $8, $8, 1

b while

all\_done: li $10, '\n' # print newline character

putc $10

done

**A program example that reads characters that form an integer, and then prints them back out:**

# a MAL program to print out the ? of a user-entered integer.

.data

# prompts

str1: .asciiz "Enter an integer: "

str2: .asciiz "The result is "

str\_error: .asciiz "\nInput error detected. Quitting.\n"

newline: .byte '\n'

# variables

int\_array: .word 0:20 # array to hold integer for printing

.text

\_\_start: la $8, str1 # print prompt

puts $8

lb $10, newline # read characters and calculate

li $11, 57 # the integer represented

li $12, 48

getc $9

get\_chars: beq $9, $10, got\_int # newline char terminates loop

bgt $9, $11, int\_error

blt $9, $12, int\_error

sub $13, $9, 48 # convert char to digit

mul $14, $14, 10 # int = int \* 10 + digit

add $14, $14, $13

getc $9

b get\_chars

int\_error: la $8, str\_error

puts $8

j end\_program

got\_int:

# $14 -- the integer to be printed

# $15 -- base address of array holding the integer

# $16 -- running address of array element

# $17 -- single digit of the integer

# $18 -- single character of the integer

print\_int: la $8, str2

puts $8

la $15, int\_array

move $16, $15

more\_digits: rem $17, $14, 10

sw $17, ($16)

add $16, $16, 4

div $14, $14, 10

bgtz $14, more\_digits

sub $16, $16, 4

bge $16, $15 more\_chars # test for result = 0

putc '0'

putc $10 # print newline

more\_chars: lw $18, ($16)

add $18, $18, 48

putc $18

sub $16, $16, 4

bge $16, $15, more\_chars

end\_program: putc $10

done

**Style conventions**

Assembly language source code tends to follow style conventions. Follow these:

* Labels start at the leftmost column.
* The mnemonics for the instructions line up at a specific column.
* Do *not* indent to show logic or blocks.
* Total number of characters on a line never exceeds 80.
* Inline comments can start at a specific column, or be to the right of the code they describe. The tradeoff: putting comments to the right better identifies the code being described. However, it is difficult to keep the comments up to date, when the code is still being debugged and changed. Placing comments on their own lines (not to the right) tends to make the code longer, and sometimes more difficult to read, but is easier to do while still debugging code.

|  |
| --- |
| Copyright © Karen Miller, 2006 |

**MAL Register Usage Conventions**

|  |  |  |
| --- | --- | --- |
| **Register Name** | **Alternate Name** | **Use** |
| $0 | $zero | constant value 0 |
| $1 | $at | reserved by the assembler |
| $2 - $3 | $v0 - $v1 | expression evaluation and subprogram return value |
| $4 - $7 | $a0 - $a3 | the first four parameters - not preserved across calls |
| $8 - $15 | $t0 - $t7 | temporaries - not preserved across calls |
| $16 - $23 | $s0 - $s7 | saved values - preserved across calls |
| $24 - $25 | $t8 - $t9 | temporaries - not preserved across calls |
| $26 - $27 | $k0 - $k1 | reserved by the operating system |
| $28 | $gp | global pointer |
| $29 | $sp | stack pointer |
| $30 | $s8 | saved value - preserved across calls |
| $31 | $ra | return address |
| $f0 - $f2 |  | floating point subprogram return value |
| $f4 - $f10 |  | temporaries - not preserved across calls |
| $f12 - $f14 |  | the first 2 floating point parameters - not preserved across calls |
| $f16 - $f18 |  | temporaries - not preserved across calls |
| $f20 - $f30 |  | saved values - preserved across calls |

Register $0 is special in that it always has contents with value 0. If it is specified as a destination operand of an instruction then the result of the instruction is not stored.

The designations "preserved across calls" and "not preserved across calls" have significant implications for both caller and callee.

"Preserved across calls" means that the caller can count on the saved value registers having the same contents before and after a subprogram call. If the callee uses the saved value registers, the callee should take measures to save the register values and restore them before returning.

"Not preserved across calls" means that the caller cannot count on the temporary registers having the same contents before and after a subprogram call. Thus the the callee can use the temporary registers freely.

For simple programs, it is best to use memory or saved value registers for the main program variables and temporaries for subprogram local variables. For more complex programs, some subprograms will call other subprograms. Then great care is needed in using registers. The runtime stack can be used as a safe storage place for data when deeply nested subprogram calls occur. Use of the runtime stack is essential for recursive subprograms.

**MIPS Subprograms**

The primary purpose for the register use conventions described above is to simplify the design of code with subprograms. In MIPS assembly language, subprograms are called by using the jal instruction. This instruction instruction has a single operand, which is the label for starting instruction in the subprogram.

The jal works like a branch instruction, with one additional feature - it saves the PC in the return address register $ra. The save is done after the PC is incremented so that $ra contains the address of the instruction that follows the jal instruction. When the subprogram is ready to return, it can use the jr instruction, specifying $ra as a jump address. Then instruction execution resumes where it left off before the call.

Most subprograms have parameters and many have returned values. In addition, subprograms use registers, creating the possibility of interfering with the callers use of registers. In order for subprograms to work, the calling code and the called subprogram must have a shared convention for the use of registers and dealing with parameters and returned values. The next section describes a relatively simple convention that is powerful enough to support recursion.

**Recommended Calling Convention for Simple Functions**

The convention described below works well for most subprograms with four or fewer parameters. For recursive subprograms with more than four parameters, it is easier to pass all parameters on the runtime stack. Fortunately, this is a rare situation.

* Use registers $a0 - $a3 for passing parameters. These caller puts values into these registers prior to calling the subprogram with jal.
* Use registers $v0 and $v1 for returned values. These caller takes values from these registers after the call.
* If you are about to call a subprogram, and a temporary register, parameter register, or the return address register contains a value that you will need after the call then save (push) the value onto the runtime stack and restore (pop) it after the call. If you are writing code for a subprogram that is calling another subprogram then you will surely need the return address register, so it should always be saved and restored.
* If you are writing code for a subprogram that needs to use a saved value register then save (push) its value at the beginning of the subprogram code and restore (pop) it before returning.

To push data onto the runtime stack use the following code:

sw *register\_name*, 0($sp)

sub $sp, $sp, 4

To pop data from the runtime stack use the following code:

add $sp, $sp, 4

lw *register\_name*, 0($sp)

Always use word loads and stores for pushes and pops, and always change $sp by multiples of 4. Otherwise, you will end up with unaligned memory references. Also take great care to ensure that no matter how the subprogram code is executed, it always does the same number of pushes and pops, and that pops are done in reverse order to the pushes. Otherwise, you will end with confusion. The worst possibility is that the wrong value gets popped into $ra.

For an example of this register usage convention, see [fact.s: An Example of a MAL Subprogram](http://www.d.umn.edu/%7Egshute/spimsal/fact.html).