Going From C to MIPS Assembly

Basic Operations: Loops, Conditionals

Charles Gordon

(Version 1.1, September 2000)

1 Overview

At this point in the course, you should be reasonably familiar with the basic concepts of MIPS assembly. This includes registers, instruction formats, addressing, and basic arithmetic and load/store operations. It might be good to review the first four sections of chapter 3 if you are still uncomfortable with these concepts. This guide will focus on the next step in learning assembly: conditional statements and loops.

2 If/Then/Else Statements

A generic if/then/else construct from C is given in figure 2.1. It consists of a conditional check followed by two possible code blocks. One of these (the then block) is executed if the conditional is true, and the other (the else block) is executed if it is false.

```
if( conditional ) {
     then block
} else {
     else block
}
```

Figure 2.1: C if construct

The assembly language version of figure 2.1 is given in figure 2.2. Notice that the conditional is the same, but the order of the else and then blocks has been reversed. This is because the conditional statement is also a branch command. If the condition is met the code branches to the then block, otherwise it continues on to the else block.

```
conditional
else block
j end
then:
then block
end:
```

Figure 2.2: Assembly if construct

More concrete examples will be given in the last section of this guide, but here is a way to think about the program flow. The conditional statement is first tested and, if it is false, program flow continues into the else block. Once the else block has finished, the jump instruction skips the then block and goes straight to the end of the if/then/else construct. If the conditional evaluates to true the else block is skipped and the then block is executed. The only tricky parts for a C programmer are remembering the jump after the else block and remembering to reverse the order of the then and else blocks.

Figure 2.3 gives a list of the C conditional operators and their corresponding assembly language instructions. Assume that a is stored in register t0 and b is in register t1. Note that all of these instructions with the exception of beq and bne are *psuedo-instructions*. They are created with a combination of beq, bne and slti. See your text for more information.

C Conditional Operator	**MIPS Assembly Instruction**
a == b	beq $t0, $t1, then
a != b	bne $t0, $t1, then
a < b	blt $t0, $t1, then
a > b	bgt $t0, $t1, then
a <= b	ble $t0, $t1, then
a >= b	bge $t0, $t1, then
a == 0	beqz $t0, then

Figure 2.3: C and Assembly Conditional Operators

3 Loops

There are three distinct types of loops in C: do/while, while and for. It may come as a surprise to some of you that they are all functionally identical. In other words, you can take any for loop and turn it into a while loop with a bare minimum of effort. The different forms merely capture the different uses of loops. Figure 3.1 lists a C for loop and a corresponding while loop that should make this idea clear.

```
int i;
for( i=0;i<10;i++ ) {
     loop body
}

int i = 0;
while( i < 10 ) {
     loop body
     i++;
}
```

Figure 3.1: C loop constructs

Loops of this sort are generally called counting loops and can be expressed in assembly as shown in figure 3.2.

```
li        $t0, 10        # t0 is a constant 10
li        $t1, 0         # t1 is our counter (i)
loop:
beq  $t1, $t0, end  # if t1 == 10 we are done
loop body
addi $t1, $t1, 1    # add 1 to t1
j         loop           # jump back to the top
end:
```

Figure 3.2: Simple assembly loop construct

No self-respecting compiler would ever produce the loop in figure 3.2, but it does exactly the same thing as the loops in figure 3.1. It is left as an exercise to speed up the assembly language version (hint: it is possible to eliminate one of the branches). Otherwise, the loop structure is easy to follow. The first two lines initialize the counter (register t1) to zero and store a constant 10 in register t0. The loop label on line three gives us somewhere to jump back to when we reach the end of the loop. The fourth line tests the counter to see if we are done and if so it jumps to the end. Otherwise, execution proceeds into the loop body and then our counter is incremented. The second to last line is an unconditional jump that returns program control to the top of the loop where the counter is once again tested.

4 Example Program

Assembly language can be horribly complex and obtuse, so a good way to go about writing programs is to first write them in a high level language like C. The last two sections of this guide have given you some simple recipes for common C constructs like loops and if/then/else statements. This next section will attempt to tie it all together by giving you a mildly complicated C program and then converting it to assembly. Note that a number of things will be left out of this example for the sake of simplicity. You will learn the correct way to make procedure calls and arrays in the next guide, for now we will take some shortcuts.

4.1 String Length (strlen)

The standard C library function for determining the length of a null terminated string is strlen. The function loops through the characters in a string until it reaches a null character and then outputs the loop count. A simple C implementation might look something like the code in figure 4.1.

```
int strlen( char* string ) {
     int count = 0;

     while( *string != '\0' ) {
          string++;
          count++;
     }

     return count;
}
```

Figure 4.1: C version of strlen

For the MIPS assembly language version we will assume that count is stored in register t0 and that the address of the string is stored in register a0 (the first argument register). A possible implementation of the function is given in figure 4.2.

```
strlen:
li      $t0, 0          # initialize the count to zero
loop:
lbu     $t1, 0($a0)     # load the next character into t1
beqz    $t1, exit       # check for the null character
addi    $a0, $a0, 1     # increment the string pointer
addi    $t0, $t0, 1     # increment the count
j       loop            # return to the top of the loop
exit:
```

Figure 4.2: Assembly versions of strlen

This is a remarkably simple procedure, but it gives a good illustration of the loop construct in action. In this case, we are using a simple counting loop that terminates as soon as the character we are looking for is found. Notice that there a number of different ways that we could have done this particular function, a few of which contain even less code. The lbu instruction on line four may be unfamiliar, but it works exactly the same way as the load word instruction with which you are familiar. Rather than loading an entire word, the lbu instruction loads a single unsigned byte (the size of a C char). If characters were four bytes each we would need to use lw in line four and then add four to a0 each time through the loop instead of one.

4.2 Maximum (max)

This simple program finds and returns the maximum value in an unsorted array. The basic operation is a loop through the array with a conditional that checks the current elements magnitude against that of the current best value. A listing for the C version is given in figure 4.3.

```
int max( int* array, int size ) {
     int maximum = array[0];

     for( int i=1;i<size;i++ )
          if( array[i] > maximum )
               maximum = array[i];

     return maximum;
}
```

Figure 4.3: C version of max

For the assembly language version we will assume that the array address is in a0 and the size is in a1. The maximum value will be stored in t0 and the loop counter in t1. A listing is given in figure 4.4.

```
max:
lw   $t0, 0($a0)     # load the first array value into t0
li   $t1, 1          # initialize the counter to one
loop:
beq  $t1, $a1, exit # exit if we reach the end of the array
addi $a0, $a0, 4     # increment the pointer by one word
addi $t1, $t1, 1     # increment the loop counter
lw   $t2, 0($a0)     # store the next array value into t2
ble  $t2, $t0, end_if
move $t0, $t2        # found a new maximum, store it in t0
end_if:
j    loop            # repeat the loop
exit:
```

Figure 4.4: Assembly version of max

This program illustrates both the if construct and the loop construct. You should be starting to notice how difficult it is to keep track of variables in an assembly language program. It helps immensely when writing programs of your own to keep a small table on a piece of paper. Otherwise it is easy to get lost in the rather sparse register namespace. It is also very instructive to attempt to rewrite this program in a slightly different way (i.e. make the loop count down or have it look for the minimum value).