How to use the MIPS compiler

You're reading this document because you want to know how the MIPS compiler can make your life easier. While it cannot yet make you breakfast or create your CPU for you, it can do a number of helpful things. The MIPS compiler's main purpose in life is to turn C code into MIPS assembly. This is helpful because everyone prefers programming in a high-level language like C with such niceties as for loops as opposed to simple unstructured MIPS assembly.

However, there are a number of caveats to this task at the moment.

First off, this compiler does not allow you to include any files. So you can't include stdio.h, stdlib.h, string.h, or anything like that. You can't include headers of your own. The compiler is only going to compile code that it can see when it opens the file. You can, however, write your own functions in your file, and the compiler will be more than happy to compile those functions.

Secondly, your CPU is not suddenly going to be able to do things when you write C code that it can't when you write assembly. For example, if you try to multiply two variables together in C, the compiler will try to use the multiplication opcode, which we don't implement. However, if you write a shift add algorithm in C to do multiplication and call that function, it should work. If you do get an opcode that we haven't implemented, you should figure out how that opcode works and see if you can rewrite it into an opcode or two that you have implemented.

Finally, this compiler is a little silly about how you use its code. If you simply try to run the code as it comes out of the compiler, execution will start with your first function. So if you're going to try to do that, you should make sure that your main function is the first function in your code.

Now that you know these caveats, how should you use this compiler?

Well, for small programs that are simply testing various instructions to see if they work, this compiler is probably not going to be the most helpful thing. I expect this to be most helpful when you want to write a larger scale program (such as a search or sort algorithm). You can write a C program to do this, test it by running it as a executable on a workstation, and then remove the extra library stuff, and compile the core functions to MIPS. You will probably still have to write some of the entrance code and possibly some other glue kind of code to make this work, but the compiler should make the job of creating the core algorithm much easier.

On a final note, one last thing to be aware of when you're just using the compiler to write a function is how the compiler expects the function to be called. Probably the easiest way to figure this out is to write a dummy function that calls your function, and see how it does it. Also, this code does expect the stack pointer ($sp) to be initialized to a reasonable value, and the frame pointer ($fp) to be pointing to the same location.

Good luck with 437, and I hope this compiler can make your job somewhat easier.

Compiler Tutorial

To help you better understand how to use this compiler, I'm going to walk you through an example of how you might use it. You're more than welcome to follow along on your own command line.

Let's say that I'm going to be given an array of integers, and my job is to find if there are two of the same integer in this array. If I don't find a pair, I'll return 0. Clearly, the fastest way would be to sort this, but for the sake of argument, let's do this with two for loops. Here's how I would solve this problem writing in regular C:

```
#include <stdio.h>
int calc(int * arrayRoot, int size) {
     int i, j;
     for (i = 0; i < size; i++) {
          for (j = i + 1; j < size; j++) {
               if (*(arrayRoot + i) == *(arrayRoot + j)) {
                    return *(arrayRoot + i);
               }
          }
     }
     return 0;
}
int main(void) {
     int a[8] = {1, 3, 4, 5, 6, 8, 5, 2};
     printf("Return value is %i\n", calc(a, 8));
}
```

If I compile and run this, it tells me that "Return value is 5", which is what I want. So let's compile this to MIPS. But before we do that, we need to rewrite main. We can't use stdio.h, so let's assign the value of calc to a variable so we can use that. Here's our code ready to be compiled for MIPS:

```
int calc(int * arrayRoot, int size) {
     int i, j;
     for (i = 0; i < size; i++) {
          for (j = i + 1; j < size; j++) {
               if (*(arrayRoot + i) == *(arrayRoot + j)) {
                    return *(arrayRoot + i);
               }
          }
     }
     return 0;
}
int main(void) {
     int a[8] = {1, 3, 4, 5, 6, 8, 5, 2};
     int b = calc(a, 8);
}
```

And this is the output I get from the MIPS compiler (with some of my comments interspersed. I recommend reading this with the control flow. Reading it linearly may not make much sense):

```
    calc:
; This is the compiler setting up the variables for the function
; on the stack
; First, it adjusts the stack pointer
    addiu    $sp,$sp,-24
; First, we store the old value of the frame pointer on the stack
    sw   $fp,20($sp)
; Then, we set the frame pointer to the value of the stack ptr
    addu $fp,$sp,$0
; Then we store the arguments ($4 and $5) onto the stack
    sw   $4,24($fp)
    sw   $5,28($fp)
; Now we initialize i
    sw   $0,12($fp)
; Then we unconditionally branch to the label $L2
; Note that this instruction is not implemented by our assembler
; We'll have to rewrite this to a j, and maybe let the course
; staff know that the compiler is producing 'b' when it means 'j'
    b    $L2
    nop

; At this point, we have at least one more iteration of the outer
; loop left.  Let's load I into $2, increment it, and store that
; into j.  Seem familiar?  This is the initialization of the
; second loop
    $L7:
    lw   $2,12($fp)
    nop
    addiu    $2,$2,1
    sw   $2,8($fp)
; Now we jump to label $L3.  Again, we'll rewrite to 'j'
    b    $L3
    nop
; This is the body of the inner loop
    $L6:
; Here, we load i into $2, multiply it by 4 (because integers are
; four bytes wide), and add it to the array offset, which we
; loaded into $3
    lw   $2,12($fp)
    nop
    sll  $2,$2,2
    lw   $3,24($fp)
    nop
    addu $2,$3,$2
; Now we load the value at the address we've just calculated into
; $3
    lw   $3,0($2)
```

```
; Now we'll do roughly the same thing for the second value.
; Remember, we're doing a comparison here, so we need both values
    lw   $2,8($fp)
    nop
    sll  $2,$2,2
    lw   $4,24($fp)
    nop
    addu $2,$4,$2
    lw   $2,0($2)
; And now we have *(arrayRoot + i) in $3, and *(arrayRoot + j)
; in $2
    nop
; Here we do the check.  If they're not equal, branch to
; label $L4
    bne  $3,$2,$L4
    nop
; But if they are equal, then the compiler will go get
; *(arrayRoot + i) again.  (It could use the previously
; calculated value, but optimization is turned off to make
; the code overall more readable)
    lw   $2,12($fp)
    nop
    sll  $2,$2,2
    lw   $3,24($fp)
    nop
    addu $2,$3,$2
    lw   $2,0($2)
; So now we have the return value in $2, and we'll break to
; label $L5, which will take care of doing the actual returning
    b    $L5
    nop
; We're done with the body of the inner loop, so it's time to
; increment j, and then fall through to the termination check
    $L4:
    lw   $2,8($fp)
    nop
    addiu     $2,$2,1
    sw   $2,8($fp)
; Here we're going to do the loop termination check for the inner
; loop.  We load j into $3, and size into $2, just like before
    $L3:
    lw   $3,8($fp)
    lw   $2,28($fp)
    nop
; And then check if we're done.  If we're not, we'll branch to
; label $L6
    slt  $2,$3,$2
    bne  $2,$0,$L6
    nop
```

```
; If we are, we'll increment i, and then do the termination check
; for the outer loop
    lw   $2,12($fp)
    nop
    addiu     $2,$2,1
    sw   $2,12($fp)
; Here we're going to do the loop termination check for the outer
; for loop.  We load I into $3, and size (one of the arguments)
; into $2
    $L2:
    lw   $3,12($fp)
    lw   $2,28($fp)
    nop
; Do the comparison, and if they're not equal, then the loop's
; not done.  Branch to label $L7
    slt  $2,$3,$2
    bne  $2,$0,$L7
    nop
; Otherwise, we'll zero $2 (because that's what we're going to
; return.  We just left the outer loop)
    addu $2,$0,$0
    $L5:
; And here we exit the function, first setting the stack pointer
; to the value of the frame pointer
    addu $sp,$fp,$0
; Then loading the frame pointer with its old value
    lw   $fp,20($sp)
; And finally pushing the frame off of the stack, and returning
    addiu     $sp,$sp,24
    jr   $31
    nop

; And here's the main function
    main:
; First, we add some space to the stack, and then we'll
; initialize a.  It's a local variable, so it's on the stack
    addiu     $sp,$sp,-72
; Do the frame pointer stuff
    sw   $31,68($sp)
    sw   $fp,64($sp)
    addu $fp,$sp,$0
; Initialize the array
    ori  $2, $zero, 1
    sw   $2,28($fp)
    ori  $2, $zero, 3
    sw   $2,32($fp)
    ori  $2, $zero, 4
    sw   $2,36($fp)
    ori  $2, $zero, 5
```

```
    sw   $2,40($fp)
    ori  $2, $zero, 6
    sw   $2,44($fp)
    ori  $2, $zero, 8
    sw   $2,48($fp)
    ori  $2, $zero, 5
    sw   $2,52($fp)
    ori  $2, $zero, 2
    sw   $2,56($fp)
; Now we're going to prepare the arguments for our call to calc()
; First, we get the address of the array and load it into $4
    addiu    $2,$fp,28
    addu $4,$2,$0
; Then, we load 8 into $5
    ori  $5, $zero, 8
; Now, we call calc()
    jal  calc
    nop
; Now we're going to store the value that got returned (in $2)
; to the local variable b
    sw   $2,24($fp)
; And now we're cleaning up the function's execution
; First the frame pointer
    addu $sp,$fp,$0
    lw   $31,68($sp)
    lw   $fp,64($sp)
; Then the stack pointer
    addiu    $sp,$sp,72
; Then we return all the way
    jr   $31
    nop
; The compiler adds this to make sure that execution stops
; eventually
       halt
; This is a friendly reminder from the compiler that this will
; not run as is
Line 9: syntax error 'b'
```

The first thing to realise is that this code will not run as is (!!). Why? Well, first off the stack is not initialized. Second, execution is just going to start at the beginning, and then go into the calc function because it's there. But that's not what we want. So let's write a bit of assembly code to go at the beginning and initialize the stack for us and call the main function:

```
    org 0x0000
; initialize the stack pointer
    ori $sp, $0, 0x800
    addu $fp, $sp, $0
; call main
    jal main
    halt
```

Also we'll have to make a couple other changes. We need to remove that friendly warning at the very end about the syntax error. Also, our assembly doesn't like labels that start with "$". So we need to change them to something else. In this case, we can see that they all start with "$L", and change them all at once with a sed script, like so:

```
$ sed -i -e 's/\$L\LABEL/' test.asm
```

The "-i" tells sed to edit in-place (i.e. overwrite it after it's done editing), the -e specifies an expression to do the editing with, the first part of the expression (between '/' and '/') is the regular expression to search for, the second part of the expression (between the 2nd and 3rd '/') is what to replace it with, and "test.asm" is the file I want to edit. At this point, you should have a file that's ready to go. If you run this in sim and look in the memory dump, you can find the stack, and the value 5 stored in the local variable b. As one final exercise, let's make our own code to call the calc function. See if you can do this on your own, then compare with the code I've included below:

```
; At the beginning:
org 0x0000
; Set up the stack
    ori $sp, $0, 0x800
    addu $fp, $sp, $0
; Prepare the arguments.  The first one is the location of the
; array, and the second is the size.  We can figure this out
; from how main() calls calc()
    ori $4, $0, 0x804
    ori $5, $0, 0xB
    jal calc
    nop
; Now store the returned value (in $2) to a memory location.
    ori $8, $0, 0x900
    sw $2, 0($8)
    halt

; And then, at the end, our data:
; Make sure this isn't in a location that interferes with the
; stack.  The stack grow downward, so putting data right above
; it should work.
org 0x804
    cfw 0x1
    cf2 0x4
    cfw 0x9
    cfw 0x11
    cfw 0x1234
    cfw 0x94
    cfw 0x88
    cfw 0x3
    cfw 0x3
    cfw 0x8
    cfw 0x12
```

That concludes this tutorial. I hope it's been helpful, and I wish you much happy 437ing!