Introduction to MIPS Programming with Mars

This week's lab will parallel last week's lab. During the lab period, we want you to follow the instructions in this handout that lead you through the process of turning the odd.c program from last week's lab into an equivalent assembly language program. Then, before next week's lab, we ask you to use what you have learned to translate the prime.c program you wrote last week (or our version of that program) into assembly language.

As demonstrated in class, we want you to first convert the C version of whatever program you are working on into a file of assembly language comments. Then, you will fill in the blanks between the C code with assembly language instructions that implement the operations described by the lines in the C program.

You should begin by getting organized. You created a number of files last week. We encouraged you to create a folder to hold them. If you haven't already, you should create such a folder for last week's work and also create a new "Lab2" folder for this week's work. Put a copy of odd.c from last week in your new "Lab2" folder. This is a good opportunity to practice using the cd, mkdir, mv, rm, and cp commands.

Commenting Quickly

Given the desire to use your C code from last lab as comments for this lab, the first step is to find a way to quickly put the assembly language comment character, #, at the start of every line of your C code. The best way to do this is with sed, the Unix **s**tream **ed**itor.

- In a terminal window, use cd to make sure that you are in your new "Lab2" folder. Then type:

```
sed -e "s/^/#/" odd.c
```

sed takes its input file (odd.c) and applies an editing command (`s/^/#/`) to each line in the file sending the result to the standard output stream (your terminal in this case). The general form of the sed substitution command is "s/string to be replaced/replacement string/". The symbol `^` forces the string to be replaced to start at the beginning of a line of input. Thus, `s/^/#/` says to replace a copy of the empty string at the start of each line with a number sign. So, if you typed everything correctly, you should have seen a version of odd.c with a number sign at the start of each line.

Of course, this is not quite what you want. You don't want to see the updated text. You want to put it in a file that you can edit later to add assembly language instructions between the comments. To do this, simply:

- Redirect the standard output into a file by typing

```
sed -e "s/^/#/" odd.c > odd.asm
```

Landing on Mars

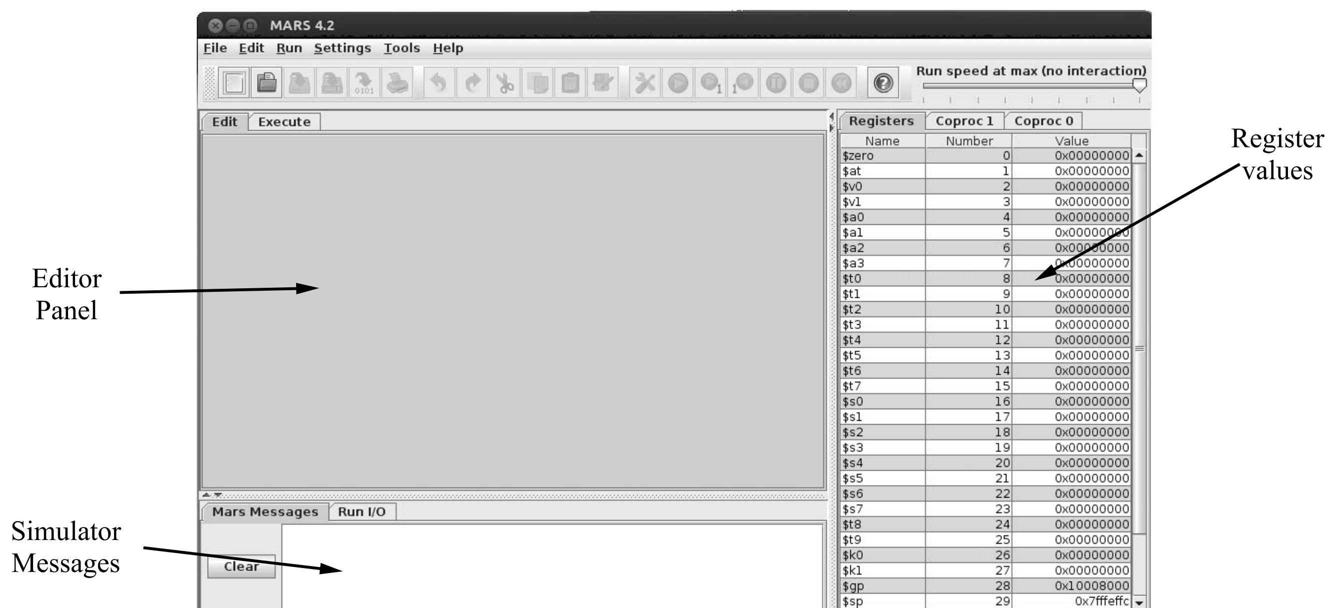
We will be using a MIPS simulator named Mars to assemble and run MIPS assembly language programs. You can (and may want to) use Emacs to edit these programs, but MIPS also includes an editor of its own. For today's lab, we will use this built-in editor.

To start the Mars program, find a terminal window and type the command

/usr/local/bin/run_mars &

(The ampersand tells Unix that the terminal window should start the program but not wait for it to complete.)

After briefly displaying a cute little picture of the red planet, the Mars program will take over your entire screen. If you like working this way, fine. Personally, I prefer to have access to multiple windows on my screen. If this is your preference too, grab the title bar of the Mars window with the mouse and pull downward. This should turn it into an independent window. You can then use the mouse to make this window a comfortable size (point the mouse at the lower right corner until a little arrow icon appears and then drag). Mars probably will not adjust its sub windows very nicely when you change the main window's size. You can fix these by pointing the mouse at the gray speckled areas between the sub windows and dragging. When you are all done, the Mars window should look something like:



The first thing you need to do is open the odd.asm file you just created with sed within the Mars editor panel.

- Select "Open" from the Mars File menu.
- Use the dialog box that appears to navigate into your Lab2 folder and open odd.asm

A Little Housekeeping

First, we will add a few bits of fairly standard code to the odd.asm file. You will want to follow roughly the same steps when you start writing any assembly language program.

MIPS memory is divided into segments. The text segment is where your machine code belongs and the data segment is where variables stored in memory (which we won't actually have yet) and string constants belong. The .text and .data assembler directives are used to specify which lines of an assembler source file belong in each segment. To keep things simple, I suggest you place all the strings you will need to define after all of your assembler code. Therefore, you only need two of these directives:

- Add a .text directive right at the start of odd.asm.

- Add a .data directive right at the end of the file. (Eventually, we will add string definitions after the directive so it will not remain at the very end.)

By default, the Mars interpreter assumes that the first line of machine code in your file is the first line that should be executed. This is inconsistent with most C programs. Because of C's limitations on forward references, function definitions usually precede the definition of main, so the first line of code to execute does not come first. Since we are going to place our code between the lines of the commented version of the C program it implements, the line where execution should begin will come after other code. To fix this:

- Point the mouse to open the Mars Settings menu and select the "Initialize Program Counter to global 'main' if defined" item.
- Right before the commented first line of main's code (the printf that displays the prompt) add a label:

  main:

  Initially, this label won't actually label anything because you have not typed in any code, but we will fix that in the next step.

Finally, a good MIPS program has to tell the simulator/operating system when it is done. To do this, the program should load 10 into $v0 and execute syscall. So, under the commented line that ends main (return 0;), add the code

```
li   $v0,10
syscall
```

At this point, you have a program that is a bit less interesting than Hello World. Since everything between where you placed the label main: and the code to terminate is a comment, the label main refers to the first line of the code to terminate. So, if you did everything correctly, the program you created should terminate as soon as it is started. It is probably a good idea to make sure it does this.

- A little to the right of the center of the row of buttons displayed near the top of the Mars window there is a button with an icon showing a wrench and a screw driver:

  This button tells Mars to assemble your code (i.e., to try to translate it into machine code). Press the button.

  If everything is correct, the text "Assemble: operation completed successfully." should appear in the messages area at the bottom of the screen and the text of your code should have been replaced by a meaningless table of numbers (probably all 0s) that describes the contents of your program's data segment and another table mixing numbers and text that shows how your code actually translated into machine code stored in the text segment. Since Mars assumes you are about to run your program it is displaying this information so that you can watch as instructions are executed and memory is modified.

- If you made a typing error somewhere, the editor panel displaying your code will remain visible and the first error will be highlighted in the window. An error message describing the problem will appear in the messages area followed by the warning "Assemble: operation completed with errors." In this case, you should examine and fix your code and then try again.
- Once the assembler succeeds, a green arrow button immediately to the right of the assemble button will become active. Click on this button to run your program. The simulator should tell you "program is finished running."
- If that was exciting enough to make you want to run your program again, you will be frustrated to find that the green arrow button is no longer active. To run a program again you either have to assemble it again or press the green double-left-arrow/rewind button. Doing either of these will re-enable the green arrow/run button.
- Once you have had enough fun running this program, you can get back to the editor by clicking on the "Edit" tab at the left edge of the Mars window underneath the row of buttons.

Input and Output

Before you finished the odd.c program last week, there was an intermediate version of the code that acted as if any number entered was odd. This version essentially ran the code:

```
int main( int argc, char * argv[] ) {
    int number;

    printf( "Enter a number:" );
    scanf( "%d", & number );

    printf( "%d is an odd number\n", number );
    return 0;
}
```

Your next step should be to get this program implemented in assembly language. You should be able to do this by simply ignoring the two lines of commented C code that precede and follow the printf that outputs the "is an odd" message. The two lines ignored should form the if statement that invokes isOdd to determine whether or not to execute the printf.

- First, since we are not ready to use real memory, you have to pick a register that will be used to hold the value of the variable number. Yes, $s0 would be a great choice. Whatever you choose, record your choice for posterity in a comment near the commented C code that declares number.
- Now, using syscall, add code to print the "Enter a number:" prompt.
  - First add an .asciiz directive to tell the assembler to place the prompt string in memory. This should go after the .data directive at the end of the program.
  - Pick a name for this message and add it as a label before the .asciiz directive.
  - Use li to place the code for printing a string in $v0 (you can look the code up by selecting "Help" from the Mars help menu and then clicking on the Syscalls tab).

  - Use la to place the address of the prompt string in $a0.
  - Finish things off with a syscall.

- Assemble the program, fix any typos, and eventually run it. Now it is as interesting as Hello World.
- In the same way, add code to print the value of the variable number followed by the "is an odd number" message. Place each sequence of assembler code you write underneath the commented lines of C code to which it corresponds.
- Finally, add code for the scanf. This will use syscall with code 5 in $v0. Be sure to move the value input from $v0 (where syscall leaves it) to the register you decided to use for number.
- Assemble and run your program. It should now echo any number you type in telling you that it is odd.

Debugging Tools

The Mars simulator includes a number of features designed to assist you when you are debugging an assembly language program. Right now, your program is probably working correctly, but we will take some time to see how the simulator's debugging features can be used to observe a program in execution. In particular, we will put a breakpoint on the last instruction of the code you wrote to implement the scanf invocation in main and then step through the next few instructions one at a time while observing how register values change.

If you have recently run your program, press the reset or assemble button so that the green arrow/run button is enabled. One of the sub-windows now displayed in the Mars window should be labeled "Text Segment". This window describes the information in the simulated machine's text segment. This is where your code is stored. The rightmost column is labeled "Source". It should contain a copy of your assembly language code with all of the comments, labels, and assembler directives removed. The next column to the left is labeled "Basic". It shows the code that is actually being executed. The main differences between the Source and Basic columns are the result of pseudo-instructions. You should notice that pseudo-instructions like LI have turned into one or more actual instructions (like ADDIU — the extra U just indicates that overflows should be ignored). Continuing to the left, the "Code" column shows the binary encoding of these instructions, and the "Address" column shows the address of the word that holds each instruction.

This brings us to the column we want to use. The "Bkpt" column is used to set breakpoints.

- Scan through the Source column to find the code for the scanf. Look for a syscall preceded by a li that puts the value 5 in $v0.
- The line after the syscall should be a move instruction that copies the value read by the syscall into the register you picked to hold the variable "number" (in the Code column it will have become an addu). Click the checkbox in the "Bkpt" column of this instruction's row to set a breakpoint on this instruction.
- Now press the green arrow button to run your program. Enter a number after the prompt appears. As soon as you do this the simulator should halt because of your breakpoint. The line where execution was suspended will be highlighted.
- Look in the list of register values displayed in the rightmost sub-window of the Mars window. Find the entry for register $v0. It should contain the number you just entered. Also look at

the entry for the register you use to hold the value of the variable number. It should still be 0.

- Now, tell the simulator to execute a single instruction by pressing the button with a smaller green arrow and the number 1. One more instruction should be executed. The value in the register that holds number should now be the same as that in $v0.

- Execute a few more instructions one at a time to see how register values change.

As we said about gdb in last week's lab, we hope that with a bit of imagination you can see how these features might come in handy while debugging.

Division

Sigh.

Now we are up to the first part of the lab that requires material we didn't quite get to in class. With this in mind, this handout will describe a few features of MIPS assembly that were not covered in class (but are in the readings!).

To enable us to tackle these features one by one, we will first modify odd.c a bit. In last week's lab, we had you write a separate isOdd function. The computation isOdd performs is so simple it is not clear it deserves to be a separate function. On the other hand, it gave you practice writing C functions. This week we will try to have it both ways. First, you will complete the program without using a separate isOdd function. Then we will implement the function and use it as you did last week.

The code from last week that used isOdd should have looked something like:

```
if ( isOdd( number ) ) {
    printf( "%d is an odd number\n", number );
}
```

If isOdd is implemented correctly, this code should be equivalent to

```
if ( number % 2 != 0 ) {
    printf( "%d is an odd number\n", number );
}
```

The first item we did not get to in class is how to evaluate the mod (%) operator. The MIPS architecture includes an instruction named DIV. It takes two register operands and computes both the quotient and remainder that result when the first is divided by the second. The tricky part is that it does not leave these two numbers in any of the usual 32 registers. Instead, the quotient ends up in a special register named LO and the remainder ends up in another special register named HI.

Luckily, to go this these special registers, there are special instructions MFLO (Move from LO), MTLO (Move to LO), MFHI (Move from HI), and MTHI (Move to HI) that move values between the 32 general purpose registers and LO and HI. For example, the code

```
DIV $s0,$s1
MFHI $t1
```

leaves $s0 % $s1 in $t1.

In class, we saw that the MIPS instruction set also includes two branch instructions name BNE (Branch not equal) and BEQ (Branch equal) that can be used to decide whether or not to branch by comparing the values of two registers. This is all you need to implement the code:

```
if ( number % 2 != 0 ) {
    printf( "%d is an odd number\n", number );
}
```

All you need to do is:

- Put a label on the first line after the code that implements the printf (probably the beginning of the code to terminate the program).
- Put code to get the value of "number % 2" before the code for the printf (remember that you can use LI to get 2 into a register).
- Put a conditional branch to the label you just added between the code to compute the mod and the code that implements the printf (you have to figure out whether to use BEQ or BNE).

Assemble and test this code.

Implementing Functions

The other material that we did not cover in class is how to implement simple functions. By "simple" we mean functions whose parameters, local variables, and return values all fit in registers. No stack space needed! The book discusses such functions on pages 325 and 326. Do not read beyond this! Not-so-simple functions start on page 327.

The techniques used to implement simple functions are straightforward. By convention, MIPS functions expect their parameter values to be in registers $a0 through $a3. So, when writing the code for such a function you should use $a0 whenever you want to refer to the first parameter, $a1 for the second and so on. Since isOdd only expects one parameter, you should write code assuming it is in $a0. It would, of course, be a good idea to state this fact in a comment right after the commented-out C function header.

In addition, a simple function like isOdd is expected to leave its result in $v0.

The really new feature involved in implementing a function like isOdd is the way you jump from the code of the main program to the code for isOdd and back again. We have seen the J (for jump) instruction in class. In a real program, a function like isOdd might be called from several locations. If we just used J to jump from each of those locations to the first line of isOdd, there would be no way to tell where the processor should jump back to when the function returned. Instead, to call a function we use the JAL (jump and link) instruction. In addition to jumping, this instruction saves the address of the instruction after the JAL in register $ra. This register's name stands for "return address."

Given that JAL is used to call a function, the last line the function executes can return to the instruction immediately after the call by using the JR (jump register) instruction in a command of the form

```
jr   $ra
```

With this background you should be able to implement isOdd. As we guide you through this, we are assuming your code for isOdd looks like the code from last week's handout:

```
\# int isOdd( int value ) {
\#          if ( value % 2 == 0 ) {
\#               return FALSE;
\#          } else {
```

```
\#              return TRUE;
\#          }
\#}
```

To implement this function:

- Place a label for the function (probably “isOdd:”) between the function’s commented-out C header and the if statement.
- Using DIV and an appropriate branch statement place code after the if statement that computes “value % 2” and branches to a label just before the second return statement if the remainder is 1.
- Place an instruction to place 0 in the register that should hold the return value ($v0) after the commented-out “return FALSE;” and follow this with a “jr $ra”.
- Place an instruction to place 1 in the register that should hold the return value ($v0) after the second commented-out return statement. Follow this with another “jr $ra”.

Now, to invoke the function, change the code for the if statement in main so that it

- Moves the value of number into the register where isOdd expects its parameter ($a0).
- Jumps to isOdd using JAL.
- Branches around the printf based on whether the value returned by isOdd in $v0 is 0 or 1.

Primes again

As mentioned in the introduction, odd.asm is just the warmup act. Once you have completed it you should begin working to convert the prime.c program you wrote last week into MIPS assembly language. As we did with odd.asm, we expect you to start by converting prime.c into a prime.asm file where the former C code serves as comments. If you don’t trust your own prime.c code, you can find a sample solution on the Labs page of the course web site.

Submitting Your work

All you have to submit this week is your prime.asm file. Make sure it contains a comment with your name in it.

You will need a terminal window to submit prime.asm. Within the terminal window you should use the cd command to make the directory containing prime.asm your current working directory. Then type the command:

```
turnin -c 237 prime.asm
```

Respond to the prompts appropriately and your code should be submitted.

Whenever you leave the lab, please remember to log out. You can do this by selecting the ”Log Out” item from the menu that appears when you depress the mouse while pointing at the little gear-like icon that appears in the upper right corner of your screen.