

CS 0447 — Spring 2017

Lab 2: Immediate Values, System Calls, and Bit Manipulation

Released: Thursday 19 January 2017

Due: Sunday 29 January 2017

Directions: Each student must submit their own solutions via CourseWeb. If you choose to work with a neighbor or partner, you must clearly include your partner's name with your submission. **Only your last submission will be graded as a complete unit, so make sure everything's there!**

For each of the three parts in this lab, you should write a separate, new program (a new file). You must name your files according to the specified format; each part will tell you the name to use. Follow instructions so that your program produces output in the correct format. For part 1, you will be asked some questions about your code. For parts 2 and 3, you will submit your programs.

Part 1: Immediate Values

Immediates are constant numbers that can be loaded into registers, used in arithmetic operations, or used in memory operations. They can often make our code a little clearer and self-explanatory by letting us use constant numbers directly in our instructions. Sometimes, they are absolutely necessary.

An example of using immediate values is seen in the following:

```
addi    $a0, $zero, 1492          # 1492 is the immediate.
# $a0 contains the year that Columbus first sailed for the Americas.

addi    $a1, $a0,    10           # 10 is the immediate.
# $a1 contains the year of Columbus's last voyage (10 years later).
```

Due to MIPS instruction set limitations, immediates are limited to being 16 bits wide. Immediates larger than 16 bits literally cannot “fit” in a single instruction (remember that instructions are always 32 bits). We often want or need to handle larger immediates (e.g., 24 bits, 32 bits). With some smart instruction manipulation, we can handle immediates of any desired size (e.g., a 32-bit immediate).

For example:

```
# 1492 and 10 both easily fit within 16 bits.
# Put the population of New York City into $a2.
addi    $a2, $zero, 8405837       # 8405837 = 0x80434D

# Problem! The population of New York City does not fit in 16 bits!
# Therefore, the above instruction is invalid.
# There is no single valid MIPS instruction that can do the above.
```

MARS will let you use immediates larger than 16 bits, even though such an immediate cannot actually fit within a single instruction! MARS is “smart enough” to automatically break your instruction that requires a 17-bit or larger immediate into multiple instructions. By running those multiple instructions, one after the other, the result will be as if MIPS could use immediates that require more than 16 bits.

Now answer questions 1 and 2 under “Lab 2” in CourseWeb:



1. What is the number of unique values that a 16-bit immediate can hold?
2. Immediates are “signed” by default, meaning that they can be either positive or negative. Given your

answer from (1), which of the following most closely represents the range of values that a 16-bit immediate can hold?

Now you try: Your goal is to write the sequence of instructions that will put the 32-bit number 0xDEADBEEF into the register \$t1. Your solution will use only immediates (and therefore, won't load any data from memory). This will require more than one instruction.

Tip: To figure out the sequence of instructions that will let you put that 32-bit number into \$t1, consider looking at how the load immediate pseudoinstruction* ("li") works. First use that pseudoinstruction to load the number 0xDEADBEEF into \$t1. Assemble and examine/test/step through the resulting program. Notice which instructions are *actually* run (pay attention to the "Basic" column). Use those instructions to put 0xDEADBEEF into \$t1 and get rid of your "li" pseudoinstruction.

There are potentially several ways to put 0xDEADBEEF into \$t1. The best way to do so will not unnecessarily disturb any registers other than \$t1.

Run the program one step at a time by pressing F7 or  as you watch the \$t1 register's value (Fig. 1) to verify how your program works. (Note: If you would like to quickly run the program to the exact point at which you want to begin stepping through your program, first set a breakpoint at that place. Then, press F5 or the Go button . MARS will execute your program until it reaches the breakpoint that you set.)

Now answer a few questions about your solution in CourseWeb:

3. What is the machine code (in hexadecimal) of the instructions in your program for part 1?
4. What instruction format are these instructions (R, I, or J)?
5. What are the values (in hexadecimal) of the immediate field in each instruction?

Your textbook's green sheet will be useful here.

Registers		Coproc 1	Coproc 0
Name	Number	Value	
\$zero	0	0x00000000	
\$at	1	0x00000000	
\$v0	2	0x00000000	
\$v1	3	0x00000000	
\$a0	4	0x00000000	
\$a1	5	0x00000000	
\$a2	6	0x00000000	
\$a3	7	0x00000000	
\$t0	8	0x00000000	
\$t1	9	0x00000000	
\$t2	10	0x00000000	
\$t3	11	0x00000000	
\$t4	12	0x00000000	

Figure 1. Where to find the value of the \$t1 register

Part 2: System Calls

* Recall that *pseudo* means false. The processor cannot and does not actually run pseudoinstructions. MARS will turn pseudoinstructions into one or more machine instructions. These machine instructions are run by the MIPS processor.

Write a MIPS program that prompts the user for two values and then prints their product in the following format: "The product of X and Y is Z", where X and Y are the values read and Z is their sum.

Sample output:

```
What is the first value?  
4  
What is the second value?  
3  
The product of 4 and 3 is 12
```

Tip: Use multiple system calls of different kinds to create the output. For example, use a print string syscall to print the first part of the output ("The product of "), then a print integer syscall, then another print string syscall (" and "), etc. Remember that "\n" is a newline (blank line). Be sure that your program's output exactly matches the format of the above sample output.

Note: For print string syscall, you need the address of the string as an argument in register \$a0. Use load address la (load address) pseudo instruction for doing this.

Here is a code snippet to display the first string for the program:

```
.data  
str: .asciiz "What is the first value?"  
.text  
la $a0, str  
addi $v0, $zero, 4  
syscall
```

mips2.asm from the lecture examples contains an example of using a syscall. You may find it useful to refer to that sample program. A complete reference of supported syscalls can be found in Help of the editor under the 'syscalls' tab. You can also find them online on MARS system calls list:

<http://courses.missouristate.edu/KenVollmar/MARS/Help/SyscallHelp.html>

Submit your file lab02part2-xxxxxx.asm in CourseWeb, where xxxxxx is your Pitt ID:

6. Submit your part 2 program. Be sure to name the file lab02part2-xxxxxx.asm.

Part 3: Bit Manipulation

Question 7: Write a program to accomplish the following:

Read an integer A from a user and store it in a register. Thirty-two bits in a word are counted from bit 0 (the least significant bit (LSB), which is the right-most bit) to bit 31 (the most significant bit (MSB), which is the left-most bit). Now, set register \$a0 to contain only bits 12, 13, 14, and 15 of A. That is, the LSB of \$a0 should contain the 12th bit of A, the second least significant bit of \$a0 should contain the 13th bit of A, etc. (You may also store the shifted bits in a register other than \$a0.)

For example, say your input integer is 1006460 which is 0x000f5b7c or
0000 0000 0000 1111 0101 1011 0111 1100 in binary. If you move bits 12, 13, 14, and 15 (which are underlined

above) to the least significant bits of `$a0`, you will get `0x00000005` in `$a0`.

Tips: Use a bitwise AND to bitmask all but the fourth least significant nibble to 0. Then move that nibble by using some sort of shift operation. See the last slide of people.cs.pitt.edu/~childers/CS0447/lectures/mips-isa2.pdf and/or Mars's help to find an appropriate shift instruction. Use syscall 34 to print `$a0` in hexadecimal. Use syscall 35 to print `$a0` in binary.

Here is a sample output from your program:

```
Please enter your integer:
1006460 <-- this is what the user inputs
Here is the input in binary: 00000000000011110101101101111100
Here is the input in hexadecimal: 0x000f5b7c
Here is the output in binary: 0000000000000000000000000000101
Here is the output in hexadecimal: 0x00000005
```

Please make sure your output lines contain the string “Here is the. . .” as shown in the sample output.

7. Submit your program “**lab02part3-<your Pitt Username>.asm**”. For example, “**lab02part3-xyz12.asm**”