

Generating & Processing the Sequence of Leonardo Numbers in *MIPS* Assembly

- **Electronic copy is due at 11:59 p.m., Sunday, October 18, 2009.**
- Late projects will not be accepted.
- Please write your last name, first name, your ID number and the project number as a comment right at the beginning of your program.
- Write comments in your code.
- You are not required to hand in a hard copy of this project.

Outline:

This is a *MIPS* program that will get you started with the *SPIM* simulator. It will also get you going writing simple *MIPS* assembly code that stores and accesses *ASCII* string data, and integer numerical word data; implements branching and loop code; and ‘walks’ along an array of integer values.

The sequence of Leonardo numbers is given by 1, 1, 3, 5, 9, 15, 25, 41, 67, 109, 177, 287, The first two elements of the sequence are initialized to $L(0) = L(1) = 1$. Each subsequent element is formed as the sum of preceding two terms plus 1:

$$L(n) = L(n-2) + L(n-1) + 1, \quad \text{for } n = 2, 3, 4, \dots .$$

Write a program which generates the sequence of Leonardo numbers, element by element, and stores the elements in memory. As it does this, the program should keep a running total of the **sum** of all elements, from the first element $L(0) = 1$ through to the element you are currently generating; it should also keep track of the “index” for that element (*i.e.*, the value of n in the expression above). Keep doing this till the sum causes an integer overflow. At that point you should stop and print out the message:

Overflow occurred when adding element k

where k gives the value of the index n for the element that caused the overflow when you added it to the running sum. This, and the other messages you output, should each appear on a line by itself (*i.e.*, the **.asciiz** message strings defined in the **.data** part of your code should terminate with the newline character **\n**).

The program should then enter a loop in which you prompt the user with the message:

Which element of the sequence do you want?

The user responds with some integer value m , and you print out the value of $L(m)$ (by retrieving it from memory where you should have previously stored it), and repeat the loop prompting the user again. This continues until the user asks you for element -1, or any other negative index, upon which the program terminates. If at any stage the user gives a (positive) index value $m > k$ (where k is the index of the last element that you generated), you should respond with the message:

Sorry! I did not get that far.

and loop back to prompt for another element.

Technical Specifications:

- Your program will need to have the messages you will be printing out defined in **.data** as **.ascii** strings.
 - You will also need to have a label **L:** in **.data** initialized with the **.word** values 1 and 1, giving the first two Leonardo numbers, $L(0)$ and $L(1)$:
 - **L: .word 1,1**
 - Your program will start generating the sequence from the third element, $L(2)$. Here are some hints to help you do that.
1. **la \$t1,L** # loads the *address* of the first element of the sequence, $L(0)$,
#into register **\$t1**.
 2. **lw \$t2,(\$t1)** # loads the *value* of $L(0)$ into register **\$t2**. Similarly,
lw \$t3,4(\$t1) # loads the value of $L(1)$ into register **\$t3**. This is because each
element in the array is an integer value stored in a word of 4 bytes.
add \$t4,\$t2,\$t3 # adds the values of $L(0)$ and $L(1)$, putting the result in register **\$t4**.
addi \$t4,\$t4,1 # adds 1 to the value in register **\$t4** and leaves the result in **\$t4**.
Note that **\$t4** now has the value of $L(2) = L(0) + L(1) + 1$.
sw \$t4,8(\$t1) # stores the value in **\$t4** into the array location associated with $L(2)$.
 3. If you now add 4 to the address of $L(0)$ which is in register **\$t1**, you will get the address of $L(1)$: **addi \$t1,\$t1,4**
and if you then repeat step 2 above you will be generating the value of $L(3) = L(1) + L(2) + 1$ and storing that value in the array location associated with $L(3)$.
 4. Obviously, you will have to write a loop that goes round steps 2 and 3 in order to continue generating further terms of the sequence. But note that in that loop:

- You have to keep track of the running sum of all elements from $L(0)$ through to the newest element you have just generated, and check that this sum does not cause an overflow (see below) before continuing with the loop.
 - You will also have to keep track of the index value n for the term of the sequence that you are generating.
- On another note, once you have built and stored the sequence in memory suppose you now want to access element m of the array, $L(m)$, $m = 0, 1, 2, \dots$. If you load the address of $L(0)$ in, say, register $\$t1$ as was done in step 1 above, then element m will be in the word that starts $4 \times m$ bytes further along from there. For example, suppose you want to access element $L(6)$:

```

la $t1,L           # load the address of L(0) into register $t1.
li $t5,6          # load the value 6 into register $t5.
sll $t5,$t5,2     # instead of explicitly multiplying the value in $t5 by 4,
                  # we achieve the same effect by shifting it left two binary
                  # places. $t5 now has the value 24 in it.
add $t5,$t1,$t5   # add the value 24 to the address of L(0) and leave the result
                  # in register $t5. $t5 now contains the address of L(6).
lw $a0,($t5)     # load the value of L(6) into register $a0.

```

Think of **L** as an array which will grow as you generate the third, fourth, fifth, elements of the sequence and store them consecutively in (word-sized) memory locations that follow the first two elements you initialized in **.data**. Thus, you will have to reserve an appropriate amount of space in **.data** to accommodate these new elements that you are generating for the array. You can do this by means of the **.space** assembler directive. In particular, consider the following:

```

L: .word 1,1     # This is the same label 'L' at which we initialize the first 2
                  # elements of the array to 1 that was mentioned above.

      .space 200  # This reserves a further 200 bytes of space immediately following
                  # the two 4-byte words with 1, 1 at label 'L' above. 200 bytes
                  # should be enough because you should find that overflow occurs
                  # when you add element L(42) of the array to the running sum.

```

You can now define more data after this if you want (for example, the **.asciiz** strings for the messages you will be printing) without fear of its getting overwritten by the expanding **L** array (assuming, of course, that your code is written correctly!).

- When you are calculating the running sum of the elements of the sequence, use the *MIPS* assembly language instruction **addu** (unsigned add), **not** the instruction **add** (signed add).

- **add** treats the integer quantities you are adding as signed 2's complement values and will cause an exception to be thrown if overflow occurs. Since you have not yet learned how to handle exceptions in *MIPS*, you should not use **add** in this program.
 - **addu** treats the integer quantities as unsigned and has the side effect that it will not throw an exception if overflow occurs.
- Even though we will be adding the terms of the sequence as if they were unsigned integers (using **addu**), we shall nevertheless treat the resulting sums as signed integers (in 2's complement representation, because computer hardware treats signed integer values as 2's complement). We can do this because, at the machine level, the values we are dealing with are essentially words composed of 32 bits, and we can view a given 32-bit quantity from any perspective that suits us (*e.g.*, an unsigned integer value, a signed integer value, a single-precision floating point value, four 1-byte ASCII characters, ...); furthermore, we can shift from one perspective to another at will. In other words, there is absolutely no data typing at the assembly language level.
 - Viewing the running sum as signed (2's complement) 32-bit integer values, the largest positive value that we can have is $2^{31}-1$. Positive values all have 0 in the leftmost bit. Overflow will have occurred when a sum has that leftmost bit 'flipped' to 1, in which case it can now be viewed as a negative 2's complement value. Thus we can test for overflow by checking if the running sum is < 0 or not, using pseudo instructions such as **bltz** (branch if < 0) or **bgez** (branch if ≥ 0).
 - **bltz** and **bgez** (and most other 'compare and branch if ...' type of instructions such as **blt**, **bge**, *etc.*) treat the quantity or quantities being tested as signed.
 - There are three 'compare and branch if ...' pseudo-instructions that treat the quantities being tested as unsigned: **bgtu** (branch if greater than, unsigned), **bleu** (branch if less than or equal to, unsigned), **bltu** (branch if less than, unsigned). Obviously, you have to be careful not to use these when testing for overflow.

Help:

- The *PCSpim* version of *SPIM* you install on your PC from the textbook CD has a *Help* feature. There is also a *SPIM* tutorial on the CD. You can also, of course, use the [MARS](#) simulator instead.
- Refer to the *HELP WITH SPIM* section of course web site (<http://www.cs.sunysb.edu/~cse220/helpwithSpim.html>).
- There are several examples in the *~cse220/examples/spim* directory on *sparky*, including *math1.asm* (covered in Lecture 10), and *length.asm* and *minmax.asm* (covered in recitation). These three are also available at the course web site as handouts to Lectures 9 & 10, but the

versions on *sparky* are source code files that you can directly copy and run on the *PCSpim* and *MARS* simulators you install on your PC.

- Section B.10 (pp. B-44 to B-80) of Appendix B of the textbook has a list of the *MIPS* instruction set. Two complementary versions of this are also available to you at the course web site:
 - http://www.cs.sunysb.edu/~cse220/Spim_inst_set.html (linked to from the *HELP WITH SPIM* section of the course web site mentioned above).
 - <http://www.cs.sunysb.edu/~cse220/Handouts/Instr.html> (which is the handout to Lecture 11).

Electronic Submission:

- From your account on *sparky*, submit an electronic copy of your program to the *sparky cse220* account.
 - You have to be logged in to your account on *sparky* which should already have a copy of the *.asm file you want to submit in the directory from which you will execute the submit command.
 - Execute: `~cse220/submit`
 - The submit command will ask you for a file name. Give the full file name, including the “.asm” extension.
 - Your project will then be copied under your name to a handin directory in the *cse220* class account.
 - If your submission is successful, you will get an acknowledgement right away.
 - If you are having problems, try typing `./<file-name>` (including the `.asm` file name extension) when asked for the file name.
 - If you submit more than once, each new submission overwrites the previous one, so only the last submission will be retained.
- You must submit electronically on *sparky*. Please do not email your program from *sparky* or anywhere else (*yahoo*, *gmail*, *etc.*) – not to me, not to the TA and certainly not to the *cse220* course account on *sparky*. (In fact, do not send email of *any* kind to the *cse220* account on *sparky* – we never check it!)
- If you are having problems with the submit command and decide not to continue with electronic submission of your project, then at least do not touch the *.asm file you have on *sparky* after the due date and time. You can always show the timestamp of the file to the TA and the TA can then copy it to the *cse220* course account.