CSE 410
Programming Project 1

Assigned: Wednesday, October 10, 2001
Due: Wednesday, October 17, 2001

Introduction

This first project includes two programs.

Adder. The first program requires that you develop a procedure to accept a string of characters representing a number, decode it, and return the actual value of the number. The main program calls your procedure to add together the two numbers provided on the command line.

Slicer. The second program requires a procedure to take a pointer to a bit string, and a center bit number, copy the 3 bits centered on the center bit, and return those bits to the caller. The main program calls your procedure to slice out the 3 bits specified on the command line.

The goal of writing these programs is to help you learn about passing arguments to programs and procedures and using the values they provide, while developing procedures that will be useful in the second and third projects.

For both programs, skeleton code is provided (on the class assignments web page) that reads arguments from the command line and sets them up for your procedure. The comment header for your procedure is also included in the skeleton. You just have to write the code to implement the procedure!

The grading for this project is as follows:

Adder program:	2 points
Hex decode extension:	1 point
Questions:	2 points
Slicer program:	2 points
Specify bit string extension:	1 point
Questions:	2 points
Total:	10 points * 5 = 50 points for the project

A Note about operating PCSpim

PCSpim does not rebuild the argument list in memory if you run it twice in a row, even though you have specified new arguments to the program. You must reload your program before each run or you will not get the results you expect!

Program: Adder

Your program will read two numbers (decimal) from its command line, and print out a message indicating their sum. For example, if the input to your program is 13 27, then your program's output should be:

13 + 27 = 40

What follows is a brief summary of how command line arguments are made available to a program when that program is first run, followed by a description of how to convert an integer from its textual representation to a representation the machine can perform arithmetic functions on.

Command Line Arguments

If you have some familiarity with C/C++ style command line processing, you will recognize that this is very similar.

When the program starts, argc (the number of arguments) is stored in register a0, while the address of argv (the array of pointers to the string arguments) is stored in a1.

For example:

$a0	argc	2	a number
$a1	argv	0x7FFFEDEC	the address of argv

Since argv is an array, $a1 actually holds a pointer to the start of the memory holding that array. In our example, argv[0] is stored at 0x7FFFEDEC. Each element of the argv array is a pointer to a null-terminated character string. So in fact, argv[0] is also an address, it is the address of the beginning of the first string argument that you entered on the command line.

argv[0] 0x7FFFEE45 the address of the beginning of the null-terminated string holding the first command line argument

In pointer terms, you must dereference the value stored in $a1 in order to get the beginning of the array of pointers to the actual data. To get the first argument you must first dereference $a1, storing that value (in $t0, for example), then you must dereference that value ($t0), giving you the start of the string.

The pointer to the second argument (argv[1]) is stored in the next word after the first argument (argv[0]). In our example, you would need to add 4 to the value in $t0 to get the next word, which you would then derefernce to get the start of the second argument.

A more pictorial description is:

$a0 argument count

$a1 pointer to start of array of argument (0x7FFFEDEC in this example)

0x7FFFEDEC	pointer to start of first argument (0x7FFFEE45 in this example)
0x7FFFEDF0	pointer to start of second argument (e.g., 0x7FFFEE49)
0x7FFFEDF4	pointer to start of third argument (e.g., 0x7FFFEE51)
...	for as many arguments as there are
0x7FFFEE45	the first character in the first argument
0x7FFFEE46	the second character in the first argument
0x7FFFEE47	the third character in the first argument
0x7FFFEE48	a zero byte, which terminates the string
0x7FFFEE49	the first character in the second argument
0x7FFFEE50	a zero byte, which terminates the string
...	

The skeleton program that we have provided contains all the argument selecting code you need for the basic assignment. If you extend the program by adding additional arguments or options, you will need to extend this part of the code to check for and obtain those command line arguments.

Converting a textual representation of a number to a machine representation

This only describes converting from a number in base 10 (decimal) to a number in base 2 (binary). In order to deal with numbers in hexidecimal, the process is very similar, and is an extension to the basic routine. Note that you can tell a number is in hex if it starts with 0x (a zero followed by an 'x')

The basic idea is to move character by character from left to right through the number keeping a running total. At each step, you must determine the decimal value of the character in question, then multiply your total by the number base (10 in the case of decimal) and add your latest digit to the total.

For example, say the character string we have been given is "324". This will be represented in memory as four consecutive bytes: '3', '2', '4', null (ie, zero).

```
Start by setting the current total to 0
LoopLabel:
See if we have a valid digit
      If the byte is zero, we are done because the string is null
      terminated.  Check for a valid digit by subtracting the
      ascii value of '0' (which is 48 decimal) from the character
      in hand.  Remember that characters are only a single byte.
      If the result of the subtraction is between 0 and 9 we are
```

```
      good to go, and we now have the numeric value of this
      character
Multiply our current total (0) by the base (10)
      (resulting in 0 the first time through the loop)
Add the result of the subtraction (3) to the total (0)
      (resulting in 3 the first time through the loop)
Go to LoopLabel
```

With our example of 324, execution would continue as follows:

We see if the next digit is valid. It is.
We multiply our total (3) by the base (10) -> 30
We add our digit (2) to the total (30) -> 32
We loop

We see if the next digit is valid. It is.
We multiply our total (32) by the base (10) -> 320
We add our digit (4) to the total (320) -> 324
We loop

We see that the next digit is not valid, and we quit - we got the number!!

Use the MIPS instruction "**`mul $dest, $src1, $src2`**" to do the multiplication.

Extension: Hex decode

In order to extend this procedure to handle hex values, you have to do a little bit of extra work.

First, if a string represents a hex number, it will start with 0x. You need to recognize those characters if they are present and set the appropriate flags or values so that you decode the number using base 16 instead of base 10.

Second, checking for a valid digit is a little more complex because the hex digits include 0 through 9 and also "A" through "F" (and also "a" through "f").

Third, the number base that you are multiplying the current total by is now 16 instead of 10, as it was when you were decoding a decimal number.

Program: Slicer

Your program will read two numbers (decimal) from its command line, and use those numbers to control a new procedure that isolates 3 bits from a bit string in memory and prints them out.

You should copy the decode_int procedure that you wrote for the adder, and include it in the source file for the slicer too.

The program takes two arguments

centerbitnumber: The number of the center bit of three bits in the bit string. 0..31

edgecontrol: When centerbitnumber specifies one of the edge bits of the entire bit string, edgecontrol determines what to substitute for the missing bit to the right or left. 0 or 1.

If you want to extend the program, you could allow for an optional third hex argument to specify the bitstring, instead of using the default string. You could also extend it so that the bit strings can be more than one word long.

A "bitstring" is just the contents of memory, considered bit by bit. For example, if the contents of memory at address 0x7FFFE040 are 0x8000000F, then the binary representation of those 32 bits is

1000 0000 0000 0000 0000 0000 0000 1111

and one would say that the 32-bit bitstring at that address starts with four 1s, followed by twenty seven 0s, followed by a single 1.

The bitSlice3 procedure that you will write for this program takes a value for the center bit, and then isolates and returns that bit and the bits on either side of it.

So, for the example word above, if the center bit was 3, then the result would be 011, and so on.

centerbit	bitstring	result
3	1000 0000 0000 0000 0000 0000 000**0 11**11	011
4	1000 0000 0000 0000 0000 0000 00**00 1**111	001
15	1000 0000 0000 000**0 00**00 0000 0000 1111	000
30	**100**0 0000 0000 0000 0000 0000 0000 1111	100

One detail that needs to be taken care of is what happens at the edges. Obviously if you specify 0 or 31 as the center bit, there is a missing bit when you are trying to select the bits on either side. The edgecontrol parameter is supplied for this situation. If edge

control is 0, you substitute a 0 for the missing bit, if edgecontrol is 1,you substitute a 1 for the missing bit.

centerbit	edge control	bitstring	result
0	0	1000 0000 0000 0000 0000 0000 0000 11**11**	110
0	1	1000 0000 0000 0000 0000 0000 0000 11**11**	111
31	0	**10**00 0000 0000 0000 0000 0000 0000 1111	010
31	1	**10**00 0000 0000 0000 0000 0000 0000 1111	110

Extension: Specify bit string

The extension for this program is to be able to specify the bit string to be used, rather than just using a value hard coded into the program as it is in the skeleton.

One relatively easy way to do this is to add an optional third parameter to the command line. This parameter allows the user to specify a bit string up to 32 bits long using a hex value. This should be a hex value, since it is extremely difficult to recognize the bit pattern that will result from specifying a decimal number in all but a few special cases.

Then apply the same bitSlice3 procedure to the user-supplied value as you did to the hard coded default value.

If you find all this too easy, there are lots of interesting extensions beyond this. You could allow bit strings longer than one word, either on the command line or in the code. You could supply edge control that uses the bit at the other end of the bitstring for the substitution instead of a fixed value. You could allow variable length slices. You could build a test harness that loops through all possible center bits for a variety of bit strings, and compares the results against a table of expected results. And so on.