**1.** In a MIPS32 program, the instruction at memory address 0x0040\_C408 is j end\_if which jumps to the

label end\_if which is at memory address 0x0041\_829C. What would be the encoding of this instruction? Express

your answer as an 8-hexdigit integer. For full credit explain how your answer was obtained.

**2.** In a MIPS32 program, the instruction at memory address 0x0040\_C4D4 is beq $t0, $t1, loop which

jumps to the label loop which is at memory address 0x0040\_6478. What would be the encoding of this instruction?

Express your answer as an 8-hexdigit integer. For full credit explain how your answer was obtained.

**3. Name your source code file hw04\_01.s** The integer square root of an integer n, isqrt(n), is defined to

be floor(sqrt(n)) where the floor of n is the largest integer that is less than or equal to n. For example,isqrt(2) = 1

since sqrt(2) is approximately 1.4142 and floor(1.4142) = 1. As another example, consider n = 67531;sqrt(67531) is

approximately 259.867, meaning that isqrt(67531) should be 259 since floor(259.867) is 259. Shown below is the

pseudocode for a fairly efficient algorithm for computing isqt(n),

**Function isqrt**(**Input: n** as **unsigned int**) **Returns unsigned int**

**unsigned int rem** ← 0, **root** ← 0

**for i** ← 0 **to** 15 **do**

***root*** = **root** leftshift 1

***rem*** = (**rem** leftshift 2) + (**n** rightshift 30)

***n*** = **n** leftshift 2

**increment root**

**if root** ≤ **rem then**

***rem*** = **rem** - **root**

**increment root**

**else**

**decrement root**

**end if**

**end for**

**return root** rightshift 1

**End Function isqrt**

This algorithm will work as long as n is not so large as to cause overflow (I have not determined the maximum value

of n). For this exercise you are to write a complete MIPS assembly language program which: (1) displays a prompt

asking the user to enter an integer n greater than or equal to 0; (2) reads a value from the keyboard into n; (3)

calculates and displays isqrt(n). Here is a sample run, where user input is in **bold**,

Enter n? **67531**

isqrt(67531) = 289

**Programming Requirements**

1. Your program shall consist of two functions: main() and isqrt(). isqrt() shall be implemented as described above.

main() shall contain the remainder of the code which prompts for n, reads n, calls isqrt() passing n, and displays

the return value from isqrt(n).

2. There shall be no global variables, i.e., the .data section shall only contain definitions for string literals. All

variables used used in main() and isqrt() shall be defined as local variables in each function's stack frame.

3. The MIPS procedure calling convention shall be followed, i.e., arguments to function calls shall be placed in $a

registers, return values from a function shall be placed in $v registers, the caller must save any $t registers that it

does not want overwritten by the callee, and the callee must save and restore any $s registers that it uses.

4. I do not want you to optimize the code in the ways we discussed in the lecture, i.e., by storing the values of

variables in registers rather than allocating room for them in the stack frame and avoiding memory accesses as

much as possible. Rather, I want you to allocate all local variables in the stack frame and read/write the values of

local variables from/to memory whenever that value is need. The primary objective of this exercise is to see if you

know how to call procedures, pass parameters, return values from a procedure, and access local variables, all while

setting up and tearing down a stack frame at the beginning and end of the procedure.

5. **Important Note:** when shifting left and shifting right, use the SLL and SRL instructions. There is a shift-right

arithmetic instruction SRA, but using it will not produce the correct result, as I learned.

**4 Download and complete the source code file hw04\_02.s from the course website.** In the

C language, strings variables are represented as one-dimensional arrays of 1-byte characters with a null character (the

character with ASCII value zero) at the end. The purpose of the null character is to indicate to functions that

manipulate C-strings where the string ends in memory; note the null character is not part of the string. For example,

consider storing the string "Bart Simpson" in a string variable named name,

char name[] ="Bart Simpson";

A diagram showing how name would be stored in memory is shown to the right. Note that the name of

an array is equivalent to the address of the first element of the array, so name is equivalent to 0x400 (we

say the address of name is 0x400). The string ends at 0x40B; the null character (which is not part of the

string) is at 0x40C. Starting at 0x40D is free memory where another variable could be allocated.

To allocate a locally defined array of characters on the stack, we have to subtract a value from $sp that

is the number of bytes in the array. For example, consider this pseudocode,

-- Returns the char at index **n** of **string**.

**Function strgetch**(**Input: string**[] as array of char; **Input: i** as int) **Returns Nothing**

**return string**[**i**]

**End Function strgetch**

**Function main**() **Returns Nothing**

**int n**; **char ch, name**[] ← "Bart Simpson"

***SysPrintStr***("Enter n? ")

***n*** ← **SysReadInt**()

***ch*** ← **strgetch**(**name**, **n**)

***SysPrintChar***(**ch**)

***SysExit***()

**End Function foo**

main() allocates three local variables: n which is 4 bytes (1 word), name which is 13 bytes; and ch which is 1 byte. It

is not a leaf procedure because although main calls strgetch(), main() never returns; we exit from main() by calling the

SysExit service call. Read the comments in the function header for main() regarding how main()'s stack frame is

allocated. Note that ch is allocated at $sp+0, name at $sp+1 to $sp+14, and n at $sp+16. The byte at $sp+15 is

wasted memory which is required to ensure that n is word-aligned.

In C when an array is passed as a function argument, what is passed is the address of the array. This is done because

passing the array by-value would require copying all of the elements of the array from one block of memory to another

block of memory. This takes time and wastes memory. Passing the address of an array is faster because the address is

simply a word which can be passed in a register; it also consumes less memory because there is only one version of the

array in memory. The callee can retrieve the address from the register and perform calculations to access the

individual elements of the array

The general formula for accessing the ith element of a one-dimensional array a where each element is of type t is,

&a[i] = &a + i · sizeof(t)

where the sizeof operator returns the number of bytes allocated for a value of type t. For a character array (t = char),

sizeof(char) = 1 byte. If a were an array of integers (t = int), sizeof(int) = 4 bytes in MIPS32.

strgetch() allocates two local variables: string which is 4 bytes (1 word) and contains the address of thename

argument passed from main() and i which is a 4-byte word and contains the value of the n argumentpassed from

main(). strgetch() is a leaf procedure as it does not call any other procedures, so it does not need to save and restore

$ra in the stack frame. Therefore, strgetch()'s stack frame will consist of 8 bytes (2 words).

Within strgetch(), to access string[i] we have to: (1) load string into a register; remember, string is the address of the

name argument from main(); within the stack frame, the value of string will be at $sp+4; (2) load i into a register; i is

at $sp+0 (3) add those two registers to compute &string[i]; and (4) use a lbu (load byte unsigned) to readstring[i] (or

use a sb to write a character to string[i]).

h04\_02.s contains the implementation of main() and strgetch(). Your job is to read the comments that describe the

strcpy()—string copy—function and write the assembly language code to implement that function. Documentation on

the C strcpy() function can be found at 1.

When studying the code, you will notice that I made no great effort to optimize ot and neither should you. There is

great benefit to learning how to write highly optimized assembly language code, but in order to learn the basics of

assembly language programming, it is less confusing when you do not attempt to optimize the code.