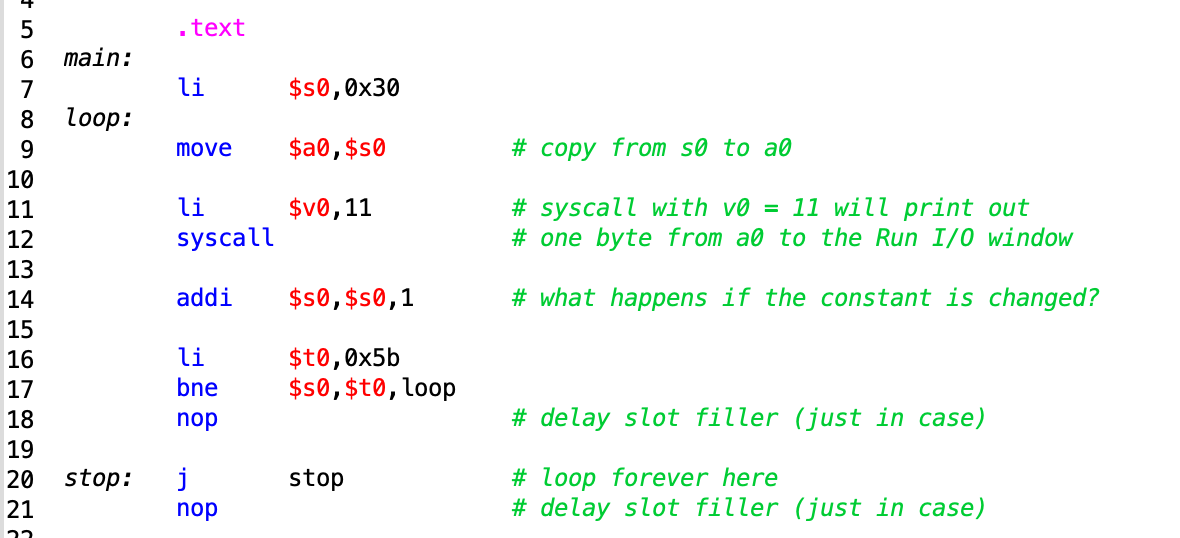
Laboratory 1 tasks

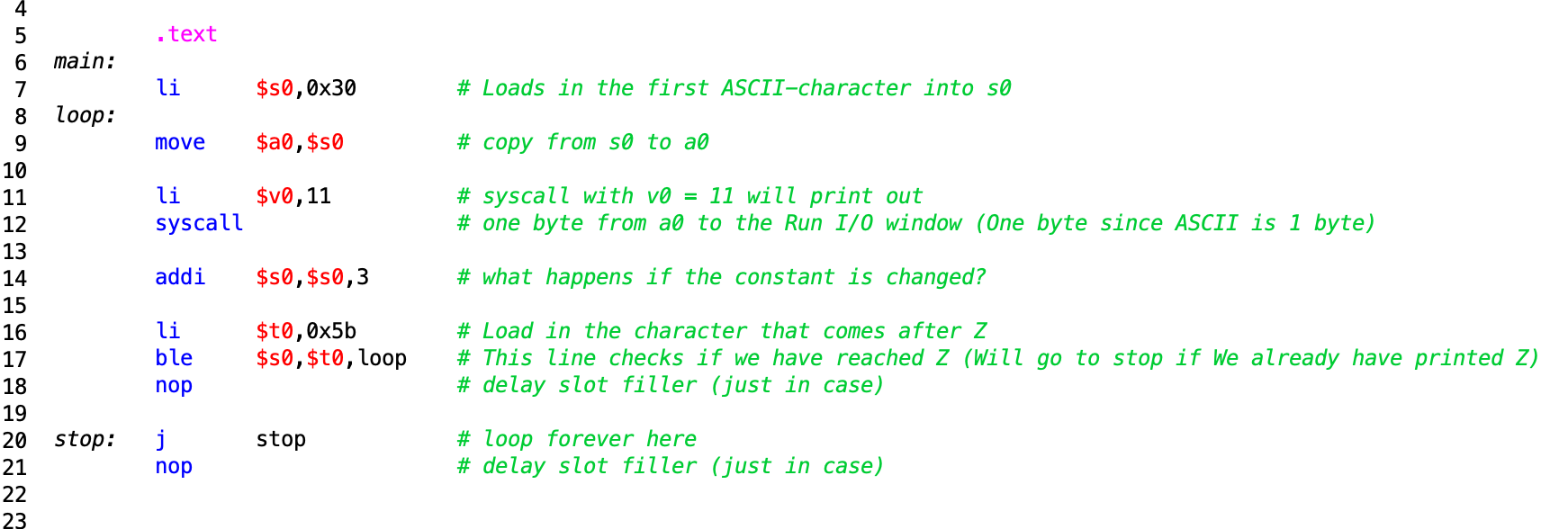
Alexander Lundqvist

**Task 1**

Which lines of code had to be changed? Why?

This is the starting code of analyze.asm. It prints out all the ASCII-characters from 0-Z in the run I/O console. The task is to change the code so that only each third character is printed.

ASCII-characters take up one byte (8 bits or 2 hex numbers). The hex range is from 0x00000030 to 0x0000005A. To produce the output specified in the task, we will change line 14 and line 17.

This is because in line 14, we previously added only one, and therefore incremented the ASCII-character to the next. (0x30 = 0, 0x31 = 1,…, 0x59 = Y, 0x5A = Z). So to get the each third ASCII-character we add 3 instead.

The previous line at 17 checked if the character was equal to the character after Z, which would signal a stop. But by taking each third we would actually skip the instance that would signal a stop and the program would loop forever. By instead changing it to ble instead, we will check if the current char is less than our “stop” char ( 0x5B = [ ).

**Task 2**

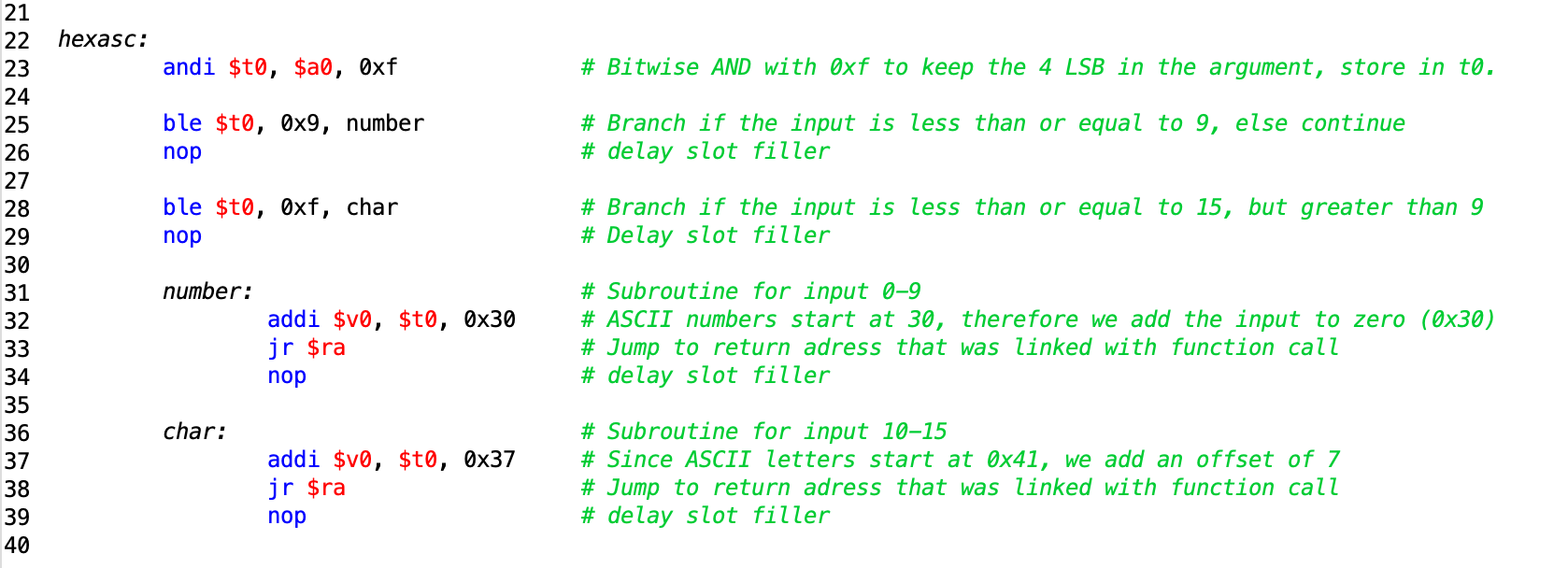
Note to teachers and students: No s-registers may be used, and no registers should be saved.

Your subroutine hexasc is called with an integer-value as an argument in register $a0, and

returns a return-value in register $v0. If the argument is 17, what is the return-value? Why?

If your solution contains a conditional-branch instruction: which input values cause the

instruction to actually branch to another location? This is called a taken branch.



To keep the last for bits (4 LSB), we perform a immediate and operation on our “input” register $a0 with the value 0x0000000F, since bitwise AND operations keep the value of a bit if it is ANDed with a one, and discard the value if it is a zero.

If we input anything higher than 15, we will get wrong answer. 17 for example will become 1. This is because we are only considering the 4 LSB in the input. And because 17 is 11 in hex, we get 1. 76 (0100**1100**) would become **C.**

We use conditional branching (ble) to determine where to continue the program. If the “input” is less than or equal to 9, we go to the label number. If it is less than or equal to 15, we jump to label char.

**Task 3**

Note to teachers and students: check register-usage and saving/restoring carefully.

Which registers are saved and restored by your subroutine? Why?

Which registers are used but not saved? Why are these not saved?

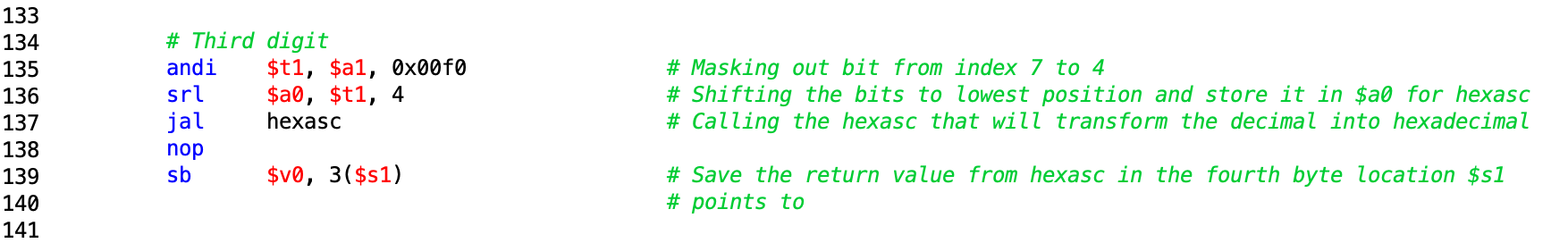
Assume the time is 16:53. Which lines of your code handle the '5'?

We push $s1 to the stack, since it is common practice that preserved registers such as $s0-$s7, need to be saved by the called function. These are also called callee-saved. In our program it doesn’t seem necessary, but we chose to save it anyway.

We also store the return address ($ra) on the stack for safekeeping since it is common practice and it is a preserved register.

The registers that are not saved on the stack are the temporary value registers ($t0-$t7). This is because they are only needed temporary inside the subroutine. They are considered caller-saved (or nonpreserved), so any subroutine that calls upon another (The callee) needs to preserve these registers in case they are needed.

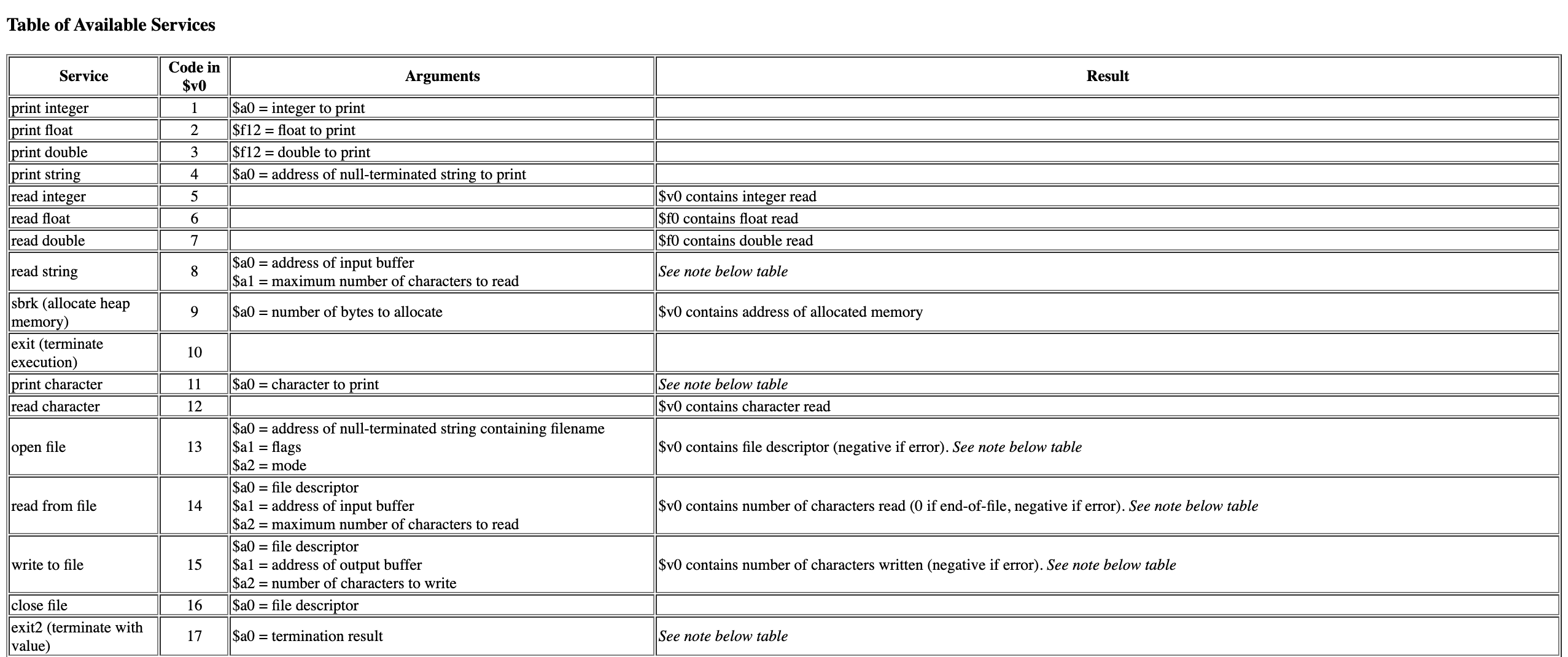
In our program, they aren’t preserved in the prewritten code and on inspection, it doesn’t seem that they need to either.

This is the code segment that handles the 3 digit.

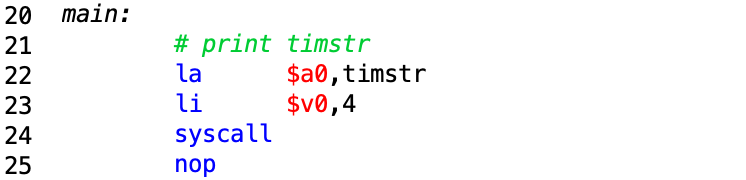
We mask out the bits that represent number 5 with the andi operation (bitwise AND keeps bits anded with 1’s and discards the ones anded with 0’s).

We then shift the bits to the lowest position and then pass it to the argument register so that hexasc can use it.

The return value from hexasc is in $v0 and we the use the sb command (Store byte) to store the output hex value in the memory pointed by $s1 (Which initially has the value of **timestr**)

Syscall uses $v0 as type of operation and $a0 as the value it performs the operation with.

Since we in the start of the program perform a syscall with operation number 4 (print string) and argument timestr (address of null-terminated string) the program will print whatever is stored in the memory pointed by timestr.

Since we through the program update what is stored in the memory location pointed by timestr, the string/time will change with each iteration of the main loop.

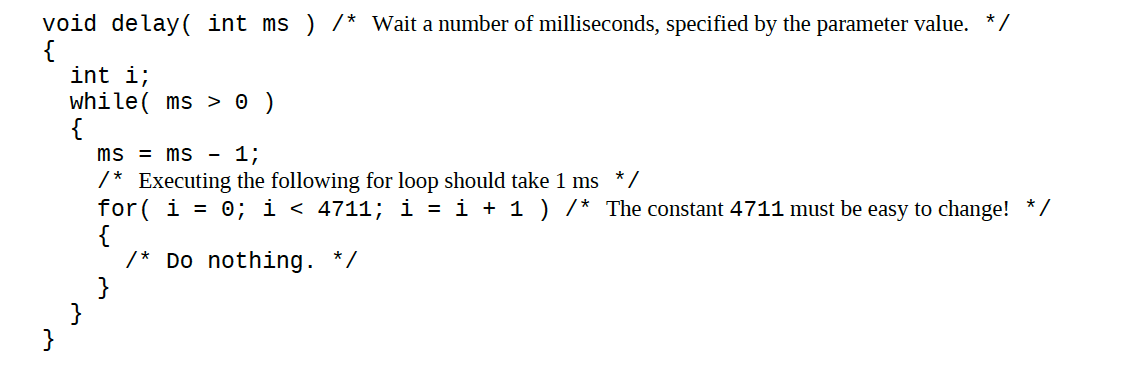
**Task 4**

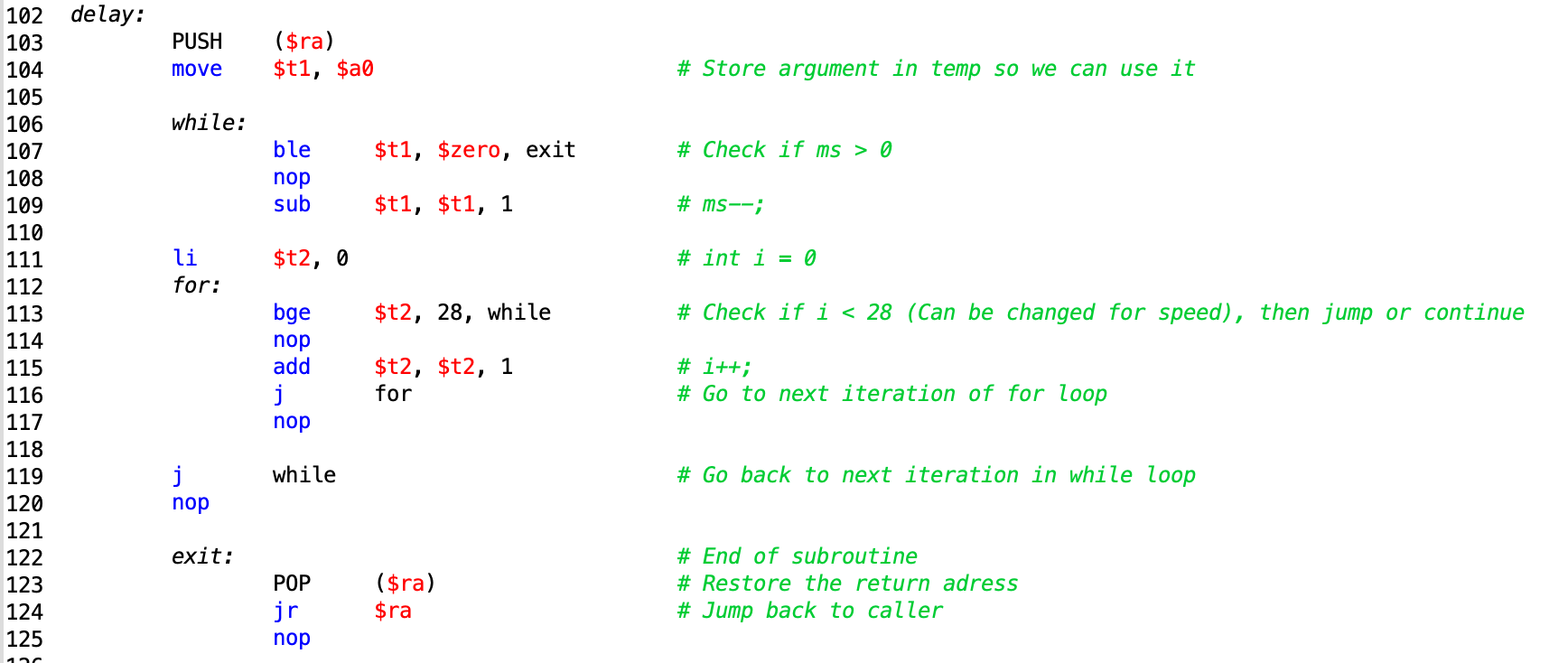
Note to teachers and students: check that the assembly code matches the C code.

If the argument value in register $a0 is zero, which instructions in your subroutine are

executed? How many times each? Why?

Repeat the previous question for a negative number: -1.

The following C-code could be written in assembly as following:

*NOTE! In the for loop there should be an addi instead of add.*

If the argument $a0 is 0, then the delay subroutine will only execute line 103, 104, 107, 123, 124 (PUSH, move, ble, POP, jr) once since we have a conditional branching that checks if the contents in $t1 (Which initially has the argument value) is larger than zero. If it is zero, that means that the condition is met and the program will jump to exit label.

Using -1 as the argument yields the same result. The program will execute very fast since it doesn’t loop in the delay subroutine at all. Ble command checks if argument is less than or equal, so both 0 and -1 are fulfill this and so the code will jump to exit label.

**Task 5**

Nothing special here.

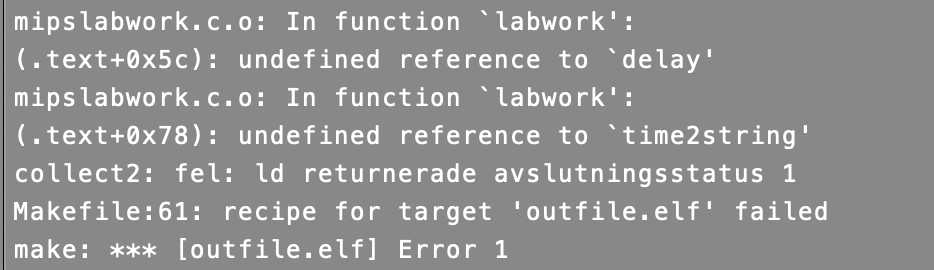
**Task 6**

What is the effect of the assembler directive .global? Why is the directive particularly

important in this assignment? The teachers will help you with this if necessary.

.global directive declares a label global, which means that other files can access this label/Refer to this label. This could be seen as “including a function in a package” just like we do in Java.

“It is for exporting symbols in your code to where it points in the object code generated. Here you mark \_start symbol global so its name is added in the object code (a.o). The linker (ld) can read that symbol in the object code and its value so it knows where to mark as an entry point in the output executable”

In this assignment where we use the make command with the MCB32Toolchain, the console will throw errors if we didn’t declare our subroutines “delay” and “time2string” as global. This is due to the fact that some other files in time4mips uses these subroutines.

*“Most large programs contain more than one file. If the programmer*

*changes only one of the files,* ***it would be wasteful to recompile and reassemble***

***the other files****. In particular, programs often call functions in*

***library files****; these library files almost never change. If a file of high-level*

*code is not changed, the associated object file need not be updated.*

*The job of the linker is to combine all of the object files into one*

*machine language file called the executable. The linker relocates the data*

*and instructions in the object files so that they are not all on top of each*

*other. It uses the information in the symbol tables* ***to adjust the addresses***

***of global variables and of labels that are relocated.****”*

*“The .data and .text keywords are assembler directives that indicate*

*where the text and data segments begin. Labels are used for global variables*

*f, g, and y. Their storage location will be determined by the assembler;*

*for now, they are left as symbols in the code.”*

**Task 7**

When you move your code from the simulator to the lab-board, you have to change the

value of the constant in the delay subroutine to get correct timing. Why?

TBD