Department of Electrical and Computer Engineering

University of Central Florida

4328 Scorpius Street

Orlando, FL 32816-2362

EEL 3801

Computer Organization and Design

Laboratory Manual

Laboratory Schedule

	Tasks	Technical Topics	Submission
Week 1 Week2	**1. Introduction to MARS**	Instruction execution flow, Assembly programming, Addressing MARS installation procedure and MIPS Assembly Instructions	**1. Y**
Week 3 Week 4	**1. Introduction to Project 1:** Practice Problems and Code for Project 1	How to use registers and characters in MIPS Assembly	**1. Y**
Week 5	**1. Project 1 (Part A Code):** Write an assembly program, which uses loops to perform multiplication.	Mathematical operations and loops in Assembly	**1. Y**
Week 6	**1. Project 1 (Part B Code):** Write an assembly program, which uses loops to perform division		**1. Y**
Week 7 Week 8	**1. Project 1 (Report)** **2. Introduction to Project 2:** Practice Problems and Code for Project 2	How to work with strings and characters in MIPS Assembly	**1. Y** **2. Y**
Week 9	**1. Project 2 (Part A Code):** Write a code which finds how many times a character is used in a given statement	How to work with strings, characters, and loops in MIPS Assembly	**1. Y**
Week 10	**1. Project 2 (Part B Code):** Write a code which finds how many times a word is used in a given statement		**1. Y**
Week 11 Week 12	**1. Project 2 (Report)** **2. Introduction to Project 3:** Practice Problems and Code for Project 3	How to use functions and procedures in MIPS assembly	**1. Y** **2. Y**
Week 13	**1.Project 3 (Part A Code):** Calculate a mathematical expression using loops and functions	Floating-point calculations in MIPS using functions. Increasing efficiency considering to Dynamic/Static Instruction Count.	**1. Y**
Week 14	**1. Project 3 (Part B Code):** Multiply two 3x3 Matrices using loops and functions	Matrix Multiplication in MIPS assembly using functions. Cache utilization observation.	**1. Y**
Week 15	**1. Project 3 (Report)**		**1. Y**

*The topics and schedule may be adjusted as necessary to maximize learning outcomes.

WEEK 1 and WEEK 2: Introduction to MARS Software

MARS, the MIPS Assembly and Runtime Simulator, will assemble and simulate the execution of MIPS assembly language programs. It can be used either from a command line or through its integrated development environment (IDE). MARS is written in Java and requires at least Release 1.5 of the J2SE Java Runtime Environment (JRE) to work. It is distributed as an executable JAR file.

1) How to install and run MARS?

Windows User:

Download and install JRE through below link:
http://javadl.sun.com/webapps/download/AutoDL?BundleId=76860

Download MARS from following link and double click the icon for Mars.jar and MARS environment will open:
http://courses.missouristate.edu/kenvollmar/mars/download.htm

Mac User:

Download and install JRE through below link:
http://www.java.com/en/download/index.jsp

Download MARS from following link and double click the icon for Mars.jar and MARS environment will open:
http://courses.missouristate.edu/kenvollmar/mars/download.htm

2) Tutorial - Basic Intro into MARS

http://www.youtube.com/watch?v=z3ltaJ5UU5I

3) Basic MARS Use

There are two main windows in MARS. The Edit window is used to create and modify your program. The Execute window is used to run and debug your program. The tabs at the top of the windows are used to switch between the two.

Part 1: Editing and Assembling

Creating a new program

Select "File => New" from the MARS menu to open a blank editor window. Enter your program. The example below shows the format of a MARS program. Select "File => Save As" to save your program to disk. The ".asm" extension is recommended. Once the file has been saved for the first time, you may select "File => Save" to save changes without having to specify the file name.

```
.data
out_string: .asciiz "\nHello, World!\n"
.text
li $v0, 4
la $a0, out_string
syscall
li $v0, 10
syscall
```

Fig 1. HelloWorld program

Opening an existing program

To open an existing program, select "File = > Open" from the Mars menu. Enter the name of the program file and click the Open button.

Assembling the program

Once the program has been created and saved to disk it may be assembled (translated into MIPS machine language). Select "Run => Assemble" from the Mars menu.

If there are no errors the Execute pane will appear, showing memory and register contents prior to execution. Click the "Edit" tab if you wish to return to the Edit pane. If there are syntax errors in your program, they will appear in the Mars messages window at the bottom of the Mars screen. Each error message contains the line and position on the line where the error occurred.

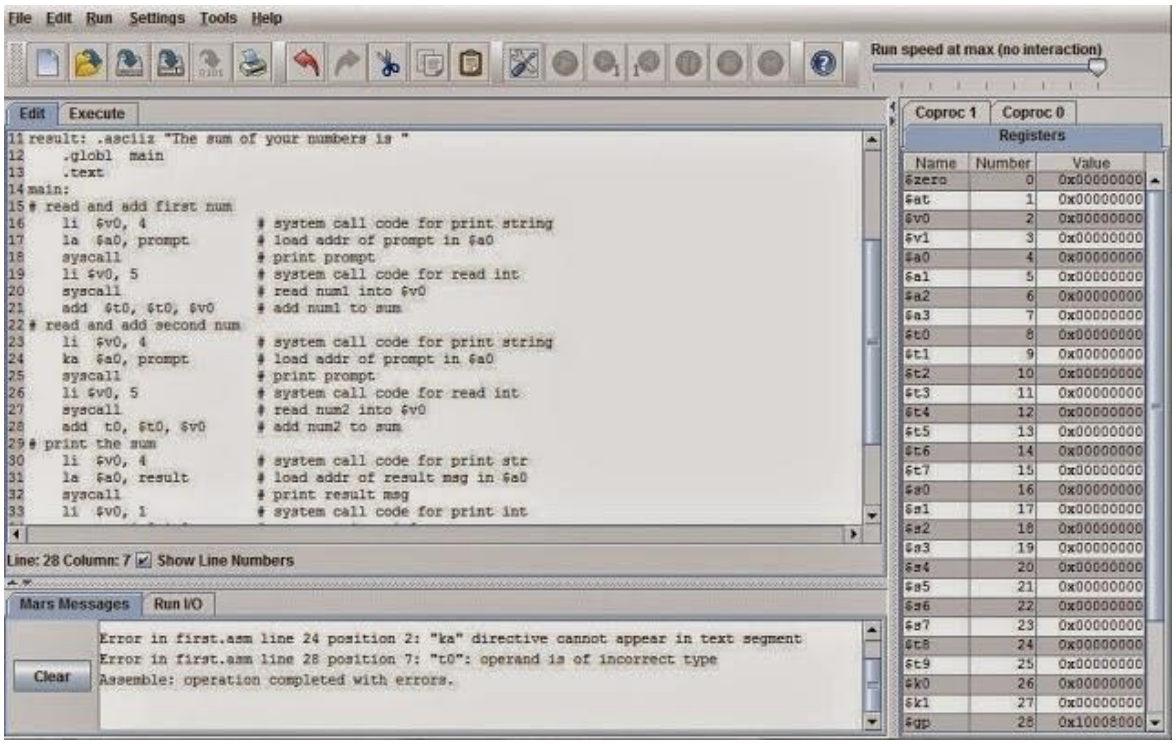


Fig 2. Errors on lines 24 and 28 shown in Mars messages window

Part 2: Executing Your Program

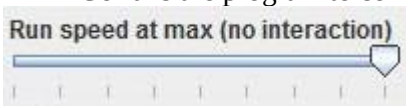
If there are no syntax errors when you assemble, the display will switch to the execute window. The execute window contains panes for the text segment and the data segment (which holds the variables) as shown in Fig. 3.

The **Text Segment** window displays the instructions. Each line contains 5 columns:

- Column 1 displays a checkbox for setting breakpoints.
- Column 2 displays the address of an instruction in hexadecimal.
- Column 3 displays the machine encoding of the instruction in hex.
- Column 4 is a mnemonic description of the machine instruction.
- Column 5 contains the assembly source that corresponds to the instruction.

Running the program

Once you have removed any syntax errors you can run your program. The Run menu and the toolbar contain the follow execution options:

-  Go runs the program to completion.
- 

The Run Speed Slider allows you to run the program at full speed or slow it down so you can watch the execution.
-  Step executes a single instruction.
-  Reset resets the program to its initial state, so that you can execute again from the beginning using the initial variable values.
-  Pause suspends execution at the currently executing instruction when you are running your program.
-  Stop terminates a running program.
-  Backstep "unexecutes" the last instruction when you are paused or stepping.

You can also set a breakpoint at any instruction by clicking the checkbox in front of the instruction in the text segment pane. During execution you can see which instruction is being executed (highlighted in yellow), which register was last modified (highlighted in green) and which variable was last changed (highlighted in blue). It's usually only possible to see the highlighting when you are stepping or running at less than full speed. Figure 3 shows the environment of "HelloWorld" program after completion.

1. Execute display is indicated by the highlighted tab.
2. Assembly code is displayed with its address, machine code, assembly code, and corresponding line from the source code file.
3. The values stored in Memory are directly editable.
4. The window onto the Memory display is controlled in several ways: previous/next arrows and a menu of common locations (e.g., top of stack).
5. The numeric base used for the display of data values and addresses (memory and registers) is selectable between decimal and hexadecimal.
6. The values stored in Registers are directly editable.
7. Breakpoints are set by a checkbox for each assembly instruction. These checkboxes are always displayed and available.
8. Selectable speed of execution allows the user to "watch the action" instead of the assembly program finishing directly.
9. MARS messages are displayed on the MARS Messages tab of the message area at the bottom of the screen. Runtime console input and output is handled in the Run I/O tab.

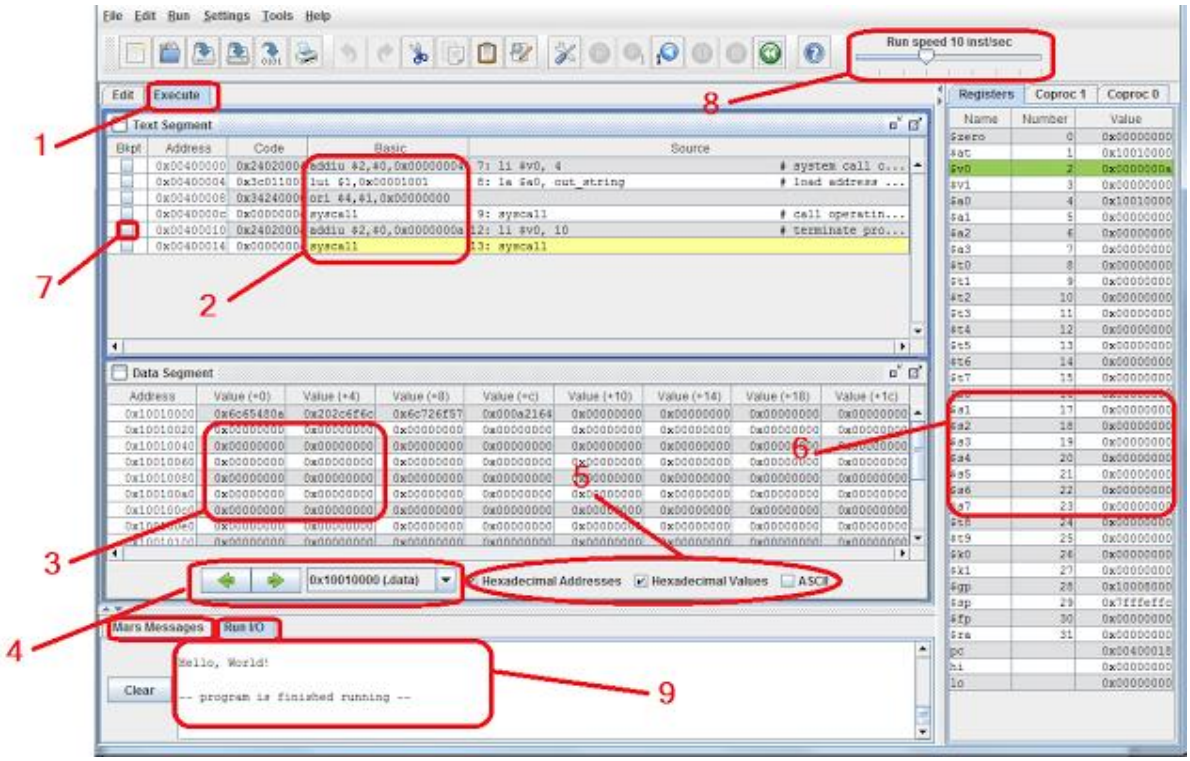


Fig 3. The environment of MARS after executing “HelloWorld” program

Closing a program

Select "File => Close" to close the current program. Always close the program before removing the program disk or exiting Mars. Exit Mars with "File => Exit.

WEEK 3 and WEEK 4: Introduction to Project 1

The Format of a MIPS Assembly Program

The components of a MIPS program are as follows:

Comments: Comments start with a **#** sign. Everything from the **#** to the end of the line is a comment.

Labels: Labels are user defined names, assigned to instructions and variables. A label is an address that starts with a letter, followed by letters and/or digits, and ends with a colon (:).

Variables: Variables are defined at the beginning of the program. The directive **`.data`** starts the variable section.

Code: The code comes after the variables. There are two directives for the code. The **`.globl`** directive specifies the external name of the function. For now, that name will be **`main`**. Later we will discuss how to create additional functions. This is followed by the **`.text`** directive, which starts the code section. The label **`main:`** comes right after the **`.text`** directive to indicate where execution should begin.

The layout of a MIPS program is as follows:

```
# comments describing the program
.data
# Subsequent items put in user data segment
.text
main:
# Subsequent items put in  user text segment
```

The **`.data`** and **`.text`** segment identifiers are required, but the **`main:`** label technically is optional.

Program Layout in the Memory: To execute a MIPS program memory must be allocated. The MIPS computer can address 4 Gigabytes of memory, from address **`0x00000000`** to **`0xffffffff`**. User memory is limited to locations below **`0x7fffffff`**. In the below figure the layout of the memory allocated to a MIPS program is shown.

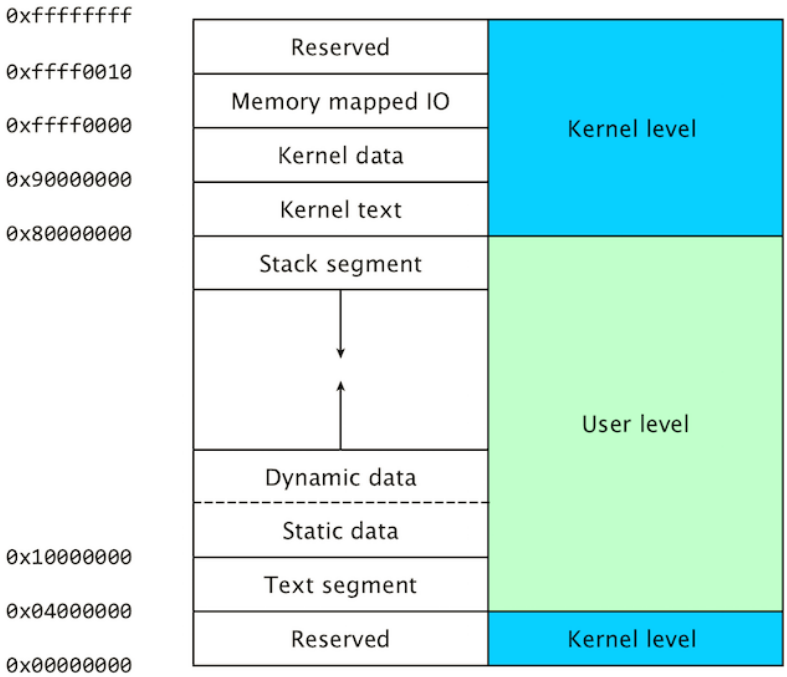


Fig.1: Program Layout in the Memory.

The purpose of the various memory segments:

- The user level code is stored in the text segment.
- Static data (data know at compile time) use by the user program is stored in the data segment.
- Dynamic data (data allocated during runtime) by the user program is stored in the heap.
- The stack is used by the user program to store temporary data during for example subroutine calls.
- Kernel level code (exception and interrupt handlers) are stored in the kernel text segment.
- Static data used by the kernel is stored in the kernel data segment.
- Memory mapped registers for IO devices are stored in the memory mapped IO segment.

Constants: are stored in the memory instead of being computed at runtime:

Character constants are enclosed in single quotes, for example 'a', 'Q', '4', '&'
Numeric constants are written in base 10, with an optional leading sign, e.g. 5, -17
String constants are enclosed in double quotes, for example "this is a string"

1. Getting Started: add.asm

To get our feet wet, we will write an assembly language program named **`add.asm`** that computes the sum of 1 and 2, and stores the result in register **`$t0`**.

1.1 Commenting

Before we start to write the executable instructions of program, however, we'll need to write a comment that describes what the program is supposed to do. In the MIPS assembly language, any text between a pound sign **#** and the subsequent newline is interpreted as a comment.

Three forms of comments are essential:

1) Program Header: this is the overall description of the program or project, which appears at the front of the code listing.
2) Code Block: comments identifying a block of code inside the program such as a loop or a subroutine.
3) Individual Instruction: one comment for each assembly language instruction is required.

Comments are vital for assembly language programs because they are notoriously difficult to read unless they are properly documented. Therefore, we start by writing the following Program Header:

```
# Your Name -- DATE
# add.asm-- A program that computes the sum of 1 and 2,
#  leaving the result in register $t0.
# Registers used:
#  t0  - holds the result.
# end of add.asm
```

Even though this program doesn't do anything yet, anyone reading our program will at least know what this program is supposed to do, and who to blame if it doesn't work. We are not finished commenting this program, but we've done all that we can do until we know a little more about how the program will work.

1.2 Finding the Right Instructions

Next, we need to figure out what instructions the computer will need to execute in order to add two numbers. It won't be long before you have memorized all of the instructions that you will need, but as you get started

with MIPS you will need to spend some time searching the lists of instructions for the right ones to use in your program. Documentation for the MIPS instruction set can be found in the textbook and in MARS as well as MIPS developers' manuals.

Luckily, as we look through the list of arithmetic instructions, we notice the **addi** instruction, which adds two numbers together. The **addi** operation takes three operands:

1. A register that will be used to store the result of the addition. For our program, this will be **$t0**.
2. A register which contains the first number to be added. We select **$t1** for this purpose and make note of this in the comments. Therefore, we're going to have to place a value of 1 into **$t1** before we can use the **addi** instruction.
3. A 16-bit constant. In this case, since 2 is a constant that easily fits in 16 bits, we can just use 2 as the third operand of **addi** .

We now know how we can add the numbers, but we have to figure out how to get the value 1 into register **$t1**. To do this, we can use the **li** (load immediate value) instruction, which loads a 16-bit constant into a register. the **li $t1, 1** is a pseudo-instruction for **addi $t1, $zero, 1** Therefore, we arrive at the following sequence of instructions:

```
# Your Name -- DATE
# add.asm-- A program that computes the sum of 1 and 2,
#  leaving the result in register $t0.
# Registers used:
#  t0 - used to hold the result.
#  t1 - used to hold the constant 1.
li $t1, 1          # load 1 into $t1.
addi $t0, $t1, 2   # $t0 = $t1 + 2.
# end of add.asm
```

The above program can be written differently as shown below:

```
# Your Name -- DATE
# add.asm-- A program that computes the sum of 1 and 2,
#  leaving the result in register $t0.
# Registers used:
#  t0 - used to hold the result.
#  t1 - used to hold the constant 1.
li $t1, 1          # load 1 into $t1.
li $t2, 2           # load 1 into $t1.
add $t0, $t1, $t2   # $t0 = $t1 + $t2.
# end of add.asm
```

1.3 Completing the Program

These two instructions perform the calculation that we want, but they do not form a complete program. Much like C, an assembly language program must contain some additional information that tells the assembler where the program begins and ends. The exact form of this information varies from assembler to assembler (note that there may be more than one assembler for a given architecture, and there are several for the MIPS architecture). This tutorial will assume that MARS is being used as the assembler and runtime environment.

In order to run the assembly program, the assembler will generate the object code information tables. This is done by reading and using directives, associate arbitrary names for labels or symbols with memory locations, produce machine language, and create an object file. Note that pseudo-instructions are instructions that assembler understands but not in machine language. In other words, to be able to perform more complicated tasks we need to employ assembler macros called pseudo-instructions. For example, **`li $t1, 1`** is a pseudo-instruction for **`addi $t1, $zero, 1`**.

1.3.1 Labels and main

To begin, we need to tell the assembler where the program starts. In MARS, program execution begins at the start of the **`.text`** segment, which can be identified with the label **`main`**. A label is a symbolic name for an address in memory. In MIPS assembly, a label is a symbol name (following the same conventions as C symbol names), followed by a colon. Labels must be the first item on a line. A location in memory may have more than one label. Therefore, to tell MARS that it should assign the label **`main`** to the first instruction of our program, we could write the following:

```
# Your Name -- DATE
# add.asm-- A program that computes the sum of 1 and 2,
#  leaving the result in register $t0.
# Registers used:
#  t0 - used to hold the result.
#  t1 - used to hold the constant 1.
main: li $t1, 1    # load 1 into $t1.
addi $t0, $t1, 2   # $t0 = $t1 + 2.
# end of add.asm
```

When a label appears alone on a line, it refers to the following memory location. Therefore, we could also write this with the label **`main`** on its own line. This is often much better style, since it allows the use of long, descriptive labels without disrupting the indentation of the program. It also leaves plenty of space on the line for the programmer to write a comment describing what the label is used for, which is very important since even relatively short assembly language programs may have a large number of labels.

Note that the MARS assembler does not permit the names of instructions to be used as labels. Therefore, a label named **`add`** is not allowed, since there is an instruction of the same name. (Of course, since the instruction names are all very short and fairly general, they don't make very descriptive label names anyway.) Giving the **`main`** label its own line (and its own comment) results in the following program:

```
# Your Name -- DATE
# add.asm-- A program that computes the sum of 1 and 2,
#  leaving the result in register $t0.
# Registers used:
#  t0 - used to hold the result.
#  t1 - used to hold the constant 1.
main:              # MARS starts execution at main.
li $t1, 1          # load 1 into $t1.
addi $t0, $t1, 2   # $t0 = $t1 + 2.
# end of add.asm
```

1.3.2 Syscalls

The end of a program is specifically defined. Similar to C, where the exit function can be called in order to halt the execution of a program, one way to halt a MIPS program is with something analogous to calling exit in C. Unlike C, however, if you forget to "call exit" your program will not gracefully exit when it reaches the end of the main function. Instead, in actual practice it may blunder on through memory, interpreting whatever it finds as instructions to execute. Generally speaking, this means that if you are lucky, your program will terminate immediately; if you are unlucky, it will do something random and then crash. The way to tell MARS that it should stop executing your program, and also to do a number of other useful things, is with a special instruction called a syscall.

The syscall instruction suspends the execution of your program and transfers control to the operating system. The operating system then looks at the contents of value register **`$v0`** to determine what it is that your program is asking it to do. Note that MARS syscalls don't actually transfer control to the operating system. Instead, they transfer control to a very simple simulated operating system that is part of the MARS program. In this case, what we want is for the operating system to do whatever is necessary to exit our program. Looking into the syscall functions available in MARS, we see that this is done by placing the value 10 (the number for the exit syscall) into **`$v0`** before executing the **`syscall`** instruction. As before, we can use the **`li`** instruction to do this:

```
# Your Name -- DATE
# add.asm-- A program that computes the sum of 1 and 2,
#  leaving the result in register $t0.
# Registers used:
#  t0 - used to hold the result.
#  t1 - used to hold the constant 1.
main:              # MARS starts execution at main.
li $t1, 1          # load 1 into $t1.
addi $t0, $t1, 2   # $t0 = $t1 + 2.
# put it into $t0.
li $v0, 10         # syscall code 10 is for exit.
syscall            # make the syscall.
# end of add.asm
```

~~~~~~

Your Exercise:

1- a) Write a program which sums two numbers specified by the user at runtime

We'll write a program named **`add2.asm`** that computes the sum of two numbers specified by the user at runtime and displays the result on the screen.
The algorithm that this program will follow is:
1. Read the two numbers from the user. We'll need two registers to hold these two numbers. We can use **`$t0`** and **`$t1`** for this.
    a. Get first number from user, put into **`$t0`**.
        i. load syscall read_int into **`$v0`**,
        ii. perform the **`syscall`**,
        iii. move the number read into **`$t0`**.
    b. Get second number from user, put into **`$t1`**
        i. load syscall read_int into **`$v0`**,
        ii. perform the **`syscall`**,
~~~~~~

iii. move the number read into **$t1**.

2. Compute the sum. We'll need a register to hold the result of this addition. We can use **$t2** for this.
3. Print the sum.
4. Exit. We already know how to do this, using syscall.

```
# Your Name -- DATE
# add2.asm-- A program that computes and prints the sum
# of two numbers specified at runtime by the user.
# Registers used:
#  $t0 - used to hold the first number.
#  $t1 - used to hold the second number.
#  $t2 - used to hold the sum of the $t1 and $t2.
#  $v0 - syscall parameter and return value.
#  $a0 - syscall parameter.
main:
 ## Get first number from user, put into $t0.
 |--------------------------------------|
 |Put your code here                    |
 |--------------------------------------|

 ## Get second number from user, put into $t1.
 |--------------------------------------|
 |Put your code here                    |
 |--------------------------------------|
  # Compute the sum.
 |--------------------------------------|
 |Put your code here                    |
 |--------------------------------------|

 # Print out $t2.
 |--------------------------------------|
 |Put your code here                    |
 |--------------------------------------|

 li $v0, 10   # syscall code 10 is for exit.
 syscall      # make the syscall.
# end of add2.asm.
```

1-b) Modify the above program to show sum of two numbers specified by the user at runtime in Hexadecimal.

~~~~~~
~~~~~~

Implementing Loops

Introduction

We'll translate a while-loop, then a for-loop. You can do the do-while loop as an exercise (and you should!). Here's a generic while-loop:

while (<cond>)
{ <while-body> }

This is how it's translated to an if-statement with goto's and a label.

L1: if (<cond>)
{ <while-body>
goto L1 ; }

Here's an example with real code:

while (i < k)
{ k++ ;
i = i * 2 ; }

Then translate it to if-statements with goto's.

L1: if (i < k)
{ k++ ;
i = i * 2 ;
goto L1 ; }

This can be straightforward to convert to MIPS. Below is some pseudocode whereby **$t1** stores i, **$t2** stores j, and **$t3** stores k. Thus, before executing in MARS you will need to substitute some real MIPS registers that you've learned instead of **$t1**, **$t2**, and **$t3**; also recall that EXIT is a label that you will need to provide. So, give it a try!

```
L1:   bge  $t1, $t2, EXIT      # branch if ! ( i < j )
      addi $t3, $t3, 1         # k++
      add  $t1, $t1, $t1       # i = i * 2
      j    L1                  # jump back to top of loop EXIT:
```

We used the pseudo-instruction **bge** for convenience. Also, rather than use the multiply instruction (which requires using two new registers, **HI** and **LO**), we multiplied by 2 by adding the value to itself.

Translating for-loops

To translate a for-loop, we'll only go part way, and translate it to if-statements and goto's. Your challenge is to do the rest on your own. Here's a generic for-loop:

for (<init> ; <cond> ; <update>)
{ <for-body> }

This is how it's translated to an if-statement with goto's.

<init>L1:
if (<inverse cond>) then goto L2:
<for-body>
<update index>
goto L1 ;
L2:

Translating nested loops

On your own, or in the next lab, you can layer the same rules for translating for-loops, and realize that the <for-body> may itself contain a for-loop, as nested loops do not need to be submitted in your portfolio today.

Summary
Translating loops is fairly straightforward, because you can translate them to if-statements with goto's. Congratulations, you've reduced the problem of translating loops to translating if-statements. Note that in assembly, sometimes we have to use the reverse logic.

~~~~~~

Your Exercise:

2-a) Write a program which increments from 0 to 15 and display the results in Decimal on the console.

2-b) Modify program above to increment from 0 to 15 and display results in Hexadecimal on the console.
~~~~~~

WEEK 5: Project 1 Part A

You are tasked to calculate a specific algebraic expansion, i.e. compute the value of f and g for the expression:

$$\boldsymbol{f = (A^4 - 4B^3 + 3C^2 - 2D)}$$

$$\boldsymbol{g = (AB^2 + C^2D^3)}$$

without using any native multiplication instructions, subroutines, and function calls. More formally, write MIPS assembly code that accepts four positive integers A, B, C, and D as input parameters. The code shall execute in MARS to prompt the user to enter four positive integers represented in decimal, each separated by the Enter key. The program shall calculate f = (A4-4B3+3C2-2D) and g = (AB2+C2D3) using your own self-written multiplication routine. The program will then output f and g in decimal and binary, using syscall for each output.

Note: To receive credit, no multiplication, no division, and no shift instructions shall be used. Namely, do not use any of {mul, mul.d, mul.s, mulo, mulou, mult, multu, mulu, div, divu, rem, sll, sllv, sra, srav, srl, srlv}. The goal to compose your own division technique. In addition, use of a loop is required for credit to realize the multiplication code. Do not use Macros, Subroutines, or Functions in this project.

Sample output for Part A is:

Enter 4 integers for A,B,C,D respectively:

15

9

21

3

f_ten = 49026

f_two = 00000000000000001011111110000010

g_ten = 13122

g_two = 00000000000000000011001101000010

The submitted program shall provide outputs in **exactly** the sequence and format shown above. To receive full credit, no additional superfluous outputs shall occur.

WEEK 6: Project 1 Part B

You are tasked to use the same positive integers from Part A to also compute:

h = f/g;

i = (f+g) MOD h_quotient;

More formally, write MIPS code to output the result of above expression of h and i without using any built-in MIPS/MARS instructions for multiplication or division. The values already entered for Part A for a, b, c, and d shall be used. Output the value of h and i in {quotient with remainder} in a format as separate decimal integers. Indicate the denominator for the remainder.

To receive credit, no multiplication, no division, and no shift instructions shall be used. Namely, do not use any of {mul, mul.d, mul.s, mulo, mulou, mult, multu, mulu, div, divu, rem, sll, sllv, sra, srav, srl, srlv}. The goal is to compose your own division technique. In addition, use of a loop is required for credit to realize the division code. It is part of the project points to design a way to realize division using a loop. **Do not use of Macro, Subroutines, or Functions in this project**.

You can refer to the definition of division and how division works. For example, given a positive integer X, and a positive integer Y where X>Y then the division X/Y is computed such that unique integers Q and R satisify X=(Y * Q + R) where $0 \leq R < Y$. The value Q is called the quotient and R is called the remainder. Some examples are:

{X = 7, Y = 2} then 7 = 2 * 3 + 1 so Q=3 and R=1

{X = 8, Y = 4} then 8 = 4 * 2 + 0 so Q=2 and R=0

{X = 13, Y = 5} then 13 = 5 * 2 + 3 so Q=2 and R=3

Sample output for Part B is:

f_ten = 49026

g_ten = 13122

h_quotient = 3

h_remainder = 9660

i_mod = 0

The submitted program shall provide outputs in **exactly** the sequence and format shown above. To receive full credit, no additional superfluous outputs shall occur.

WEEK 7 and WEEK 8: Introduction to Project 2

1) Write a C-prototype code which counts number of punctuation in the below sentence:

"This is a test. 1 2 3 4 5 !!"

Note: You are allowed to use ispunct() function. Punctuation list is provided in the image below:

2) Write a program which determines the length of a null terminated string

We'll write a program named strlength.asm that counts the number of characters in a string. In the below you can see it's C code.

```
int strlen( char* string ) {
int count = 0;
while( *string != '\0' ) {
string++;
count++;
}
return count;
}
```

In order to write the program, please follow below steps:

1. Load the address of string into $a0
2. Use $t0 as counter
3. Print $t0.
4. Exit.

```
# Your Name - DATE
# strlength.asm-- A program that determine the length of a null
terminated string
# Registers used:
# $t0 - used to hold the loop counter
# $a0 - used to hold the address of string
# $v0 - syscall parameter and return value
.data
```

```
str: .asciiz "abcde"
.text
# Load address of string into a0
|-------------------|
|Put your code here |
|-------------------|
strlen:
li $t0, 0 # initialize the count to zero
loop:
lb $t1, 0($a0) # load the next character into t1
beqz $t1, exit # check for the null character
|-------------------|
|Put your code here |
|-------------------|
exit:
li $v0, 1
move $a0, $t0
syscall
li $v0, 10
syscall
```

The lb instruction on line four may be unfamiliar, but it works exactly the same way as the load word instruction with which you are familiar. Rather than loading an entire word, the lbu instruction loads a single byte (the size of a C char).

3) Write a program which converts uppercase character to lowercase character

In this exercise, we will implement a small program that converts a string to lowercase. Something like String.toLowerCase(). The algorithm has to iterate all the string's characters (like we did in the previous post), and if the character is upper case, then we make it lower case. Characters are numbers, so we just look at their values. If a character's value is not between the character's value 'A' and the character's value 'Z', then it's not an uppercase letter. To make a character lowercase, we think in a similar way: if the characters are numbers, then mathematical operations on them are legal. Furthermore, because the character 'a' differs from the character 'A' by 32 positions, we just add to the uppercase character (say 'M') the number 32 (and we get 'm'). Let's start. The following code snippet is taken from the previous post. It's just a "while" loop with some lines of code at the end for printing on the console.

```
# Your Name - DATE
# To_lowercase.asm-- A program that convert uppercase to lowercase
# Registers used:
# That's your turn!!! Put used registers here.
.data
string: .asciiz "HeLlo WoRld" # We want to lower this string
newline: .asciiz "\n"
.text
main:
la $t0, string # Load here the string
toLowerCase:
```

```
lb $t2, 0($t0) # We do as always, get the first byte pointed by the
address
beqz $t2, end # if is equal to zero, the string is terminated
#if (character >= 'A'
|-------------------|
|Put your code here |
|-------------------|
upperCaseTest2:
# && character <= 'Z')
|-------------------|
|Put your code here |
|-------------------|
continue:
# Continue the iteration
addi $t0, $t0, 1 # Increment the address
j toLowerCase
isUpperCase:
# add 32, so it goes lower case
|-------------------|
|Put your code here |
|-------------------|
sb $t2, 0($t0) # store it in the string
j continue # continue iteration as always
end:
li $v0, 4 # Print the string
la $a0, string
syscall
li $v0, 4 # A nice newline
la $a0, newline
syscall
# We are done, exit the program
li $v0, 10
syscall
```

ASCII TABLE:

https://2.bp.blogspot.com/-nMNrhxZPNUY/UvlF8DZQnZI/AAAAAAAAAJ8/Y3PNgaOVScs/s1600/asciifull.gif

WEEK 9: Project 2 Part A

The purpose of this project is to increase your understanding of data, address, memory contents, and strings. You will be expected to apply selected MIPS assembly language instructions, assembler directives, and system calls sufficient to handle string manipulation tasks. You are tasked to find the number of selected letters present in a string, specifically the number of the occurrences of the letters K, N, I, G, H, T, and S within an input sentence, and then output the result using the format shown below.

Sample Inputs for Part A is:

coming in winners of three in a row, including back-to-back triumphs on the road, the knights returned to orlando hungry to finish out the regular season home schedule on a high note.

Sample Outputs for Part A is:

K: ---------- 3

N: ---------- 16

I: ---------- 11

G: ---------- 6

H: ---------- 12

T: ---------- 12

S: ---------- 7

K: ###

N: ################

I: ##########

G: ######

H: ############

T: ############

S: #######

The submitted program shall provide outputs in **exactly** the sequence and format shown above. To receive full credit, no additional superfluous outputs shall occur.

WEEK 10: Project 2 Part B

The purpose of this project is to increase your understanding of data, address, memory contents, and strings. You will be expected to apply selected MIPS assembly language instructions, assembler directives, and system calls sufficient enough to handle string manipulation tasks. You are tasked to develop a program that finds how many times a word is used in a given statement. To test your program, you should hardcode the below sample statement in your code and ask the user to input two different words, which are "UCF" and "KNIGHTS." Your program should not be case sensitive, it should correctly find the words regardless of how the user inputs the words, and your code should work for any words with less than 10 characters.

Sample Statement:

UCF, its athletic program, and the university's alumni and sports fans are sometimes jointly referred to as the UCF Nation, and are represented by the mascot Knightro. The Knight was chosen as the university mascot in 1970 by student election. The Knights of Pegasus was a submission put forth by students, staff, and faculty, who wished to replace UCF's original mascot, the Citronaut, which was a mix between an orange and an astronaut. The Knights were also chosen over Vincent the Vulture, which was a popular unofficial mascot among students at the time. In 1994, Knightro debuted as the Knights official athletic mascot.

Sample Output:

Please input first word: Knight (or KnIGhT, knight, …)

Please input second word: UCF (or ucf, UcF, …)

KNIGHT: 6

UCF: 3

KNIGHT: ######

UCF: ###

The submitted program shall provide outputs in **exactly** the sequence and format shown above. To receive full credit, no additional superfluous outputs shall occur.

WEEK 11 and WEEK 12: Introduction to Project 3

Subroutines in MIPS

Determines the minimum of two integers

Functions within the MIPS slides describe how one can use subroutines (also called procedures, functions, and methods) in MIPS. Because of the importance of subroutines in modern programming, most hardware designers include mechanisms to help programmers. In a high-level language like C or Java, most of the details of subroutine calling are hidden from the programmer.

MIPS has special registers to send information to and from a subroutine. The registers $a0, $a1, $a2, $a3 are used to pass arguments (or parameters) into the subroutine. The registers $v0 and $v1 are used to pass arguments (or parameters) back from the subroutine.

The stack (and stack pointer register $sp) is used for a variety of things when using subroutines. The stack is used to pass additional parameters to and from subroutines. It is also used to hold temporary values in memory that a subroutine may need. Most importantly, it is used to save the current state so the subroutine can return back to the caller once it has completed. This includes the frame pointer ($fp), the return address register ($ra), and the caller-saved registers ($s0-$s7).

Imagine you would like a program that reads two integers from the user, determine the smaller of the two, then prints the minimum value. One way to do this would be to have a subroutine that takes two arguments and returns the smaller of the two. The program shown below illustrates one way to do this.

```
## Your Name
## min_btw_2num.asm-- takes two numbers A and B
## Compare A and B
## Print out the smaller one
## Registers used:
## You can define by yourself!
.data
p1: .asciiz "Please enter the 1st integer: "
p2: .asciiz "Please enter the 2nd integer: "

.text
main:
# Get numbers from user
li $v0, 4    # Load 4=print_string into $v0
la $a0, p1   # Load address of first prompt into $a0
syscall      # Output the prompt via syscall


li $v0, 5             # Load 5=read_int into $v0
syscall               # Read an integer via syscall
add $s0, $v0, $zero # Copy from $v0 to $s0


li $v0, 4       # Load 4=print_string into $v0
la $a0, p2      # Load address of second prompt into $a0
syscall         # Output the prompt via syscall


li $v0, 5             # Load 5=read_int into $v0
```

```
syscall                # Read an integer via syscall
add $s1, $v0, $zero # Copy from $v0 to $s1


# Compute minimum
add $a0,$s0,$0     # Put argument ($s0) in $a0
add $a1,$s1,$0     # Put argument ($s1) in $a1
jal minimum        # Call minimum function, result in $v0


# Output results
add $a0, $v0, $zero # Load sum of input numbers into $a0
li $v0, 1             # Load 1=print_int into $v0
syscall               # Output the prompt via syscall


# Exit
li $v0, 10        # exit
syscall


# minimum function to compute min($a0, $a1):
|------------------|
|Put your code here|
|------------------|
return:
jr $ra    # Return to caller
```

Here are some important things to note about the above program. The **`$ra`** is the return address register which stores the address where a subroutine should return to when it is completed; it is set automatically by the **`jal`** instruction to be the address of the instruction following the **`jal`** instruction. In the subroutine, the **`jr`** instruction sets the program counter to be the address located in the **`$ra`** register.

Floating Point Calculation

This project is designed with the purpose of increasing your understanding of floating point calculation and familiarity with the floating-point instructions in MIPS. In statistics, arithmetic mean μ (or simply average) of a series a_0, a_1, …, a_{N-1} is defined as:

$$\mu = \frac{1}{N}\sum_{i=0}^{N-1} a_i$$

Write and assembly program that reads two lists of floating point numbers A and B, and displays the measures given above on the simulator's console. The program's specifications are given below:

- Each input vector should be of size 10, i.e., N=10
- The program should use PROCEDURES to compute the mean. Sample input array:

A = [0.11 0.34 1.23 5.34 0.76 0.65 0.34 0.12 0.87 0.56]

B = [7.89 6.87 9.89 7.12 6.23 8.76 8.21 7.32 7.32 8.22]

The program's output should be similar to the following:

The Mean of A = 1.03

The Mean of B = 7.78

```
## Your Name - DATE
## fp_mean.asm-- takes two numbers A and B
## Registers used:
## You can define by yourself!

.data
A: .float 0.11 0.34 1.23 5.34 0.76 0.65 0.34 0.12 0.87 0.56
B: .float 7.89 6.87 9.89 7.12 6.23 8.76 8.21 7.32 7.32 8.22
N: .float 10
meanA: .asciiz "\nThe Mean of A ="
meanB: .asciiz "\nThe Mean of B ="
.text

la $a0,N        # Load address of input vector size into $a0
l.s $f2, 0($a0) # Load the input vector size into $f2

la $a0, A       # Load the address of vector A into $a0
jal mean        # Call mean function, result in $f1

li $v0,4        # Load 4=print_string into $v0
la $a0, meanA   # Load address of first string into $a0
syscall         # Output the string via syscall

li $v0,2        # Load 2 into $v0=print single precision float
mov.s $f12,$f1  # Copy from $f1 to $f12
syscall         # output meanA


mov.s $f1,$f3  # reset $f1
la $a0, B      # Load the address of vector B into $a0
jal mean       # Call mean function, result in $f1
```

```
li $v0,4          # Load 4=print_string into $v0
la $a0, meanB     # Load address of second string into $a0
syscall           # Output the string via syscall

li $v0,2          # Load 2 into $v0=print single precision float
mov.s $f12,$f1    # Copy from $f1 to $f12
syscall           # output meanB

#Exit
li $v0,10
syscall

#mean function
mean:
|------------------|
|Put your code here|
|------------------|

 return:
|------------------|
|Put your code here|
|------------------|
```

WEEK 13: Project 3 Part A

You are tasked to calculate a specific algebraic expansion, i.e. compute the value of f and g for the expression:

$$\mathbf{f = (0.1 \times A^4) - (0.2 \times B^3) + (0.3 \times C^2) - (0.4 \times D)}$$

$$\mathbf{g = (0.1 \times AB^2) + (0.2 \times C^2D^3)}$$

Write MIPS assembly code that accepts four positive integers A, B, C, and D as input parameters. The code shall execute in MARS to prompt the user to enter four positive integers represented in decimal, each separated by the Enter key. The program shall calculate f and g using your own self-written multiplication routine. The program will then output f and g as floating-point numbers, using syscall for each output. You are tasked to increase efficiency with respect to Dynamic and Static Instruction Count. For this purposed you can use the Instruction Counter tool or Instruction Statistics tool in the MARS software.

Note: To receive credit, no multiplication, no division, and no shift instructions shall be used. Namely, none of {mul, mul.d, mul.s, mulo, mulou, mult, multu, mulu, div, divu, rem, sll, sllv, sra, srav, srl, srlv} or else a zero score will result. Thus, it is necessary to compose your own multiplication technique. In addition, use of Subroutines is required for credit to realize the multiplication code.

Sample output for Part A is:

Enter 4 integers for A, B, C, D respectively:

15

9

21

3

f = 5047.8

g = 2502.9

The submitted program shall provide outputs in **exactly** the sequence and format shown above. To receive full credit, no additional superfluous outputs shall occur.

WEEK 14: Project 3 Part B

Project Description:

In this project you will be using MARS simulator to run your assembly language program and data cache hit/miss analysis.

Consider the following segment of a C code for multiplication of two matrices A and B of size (NxN) each:

```
for (int i=0;i<N;i++) {
      for (int j=0;j<N;j++) {
            temp = 0;
            for (int k=0;k<N;k++) {
                  temp = temp + A[i][k]*B[k][j];
            }
            C[i][j] = temp;
      }
}
```

Thus, matrix C is the product of matrix A and matrix B. For example, for N=3:

$$A = \begin{bmatrix} 1 & 3 & 2 \\ 1 & 3 & 2 \\ 1 & 3 & 2 \end{bmatrix}, B = \begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 1 & 2 \end{bmatrix}$$

The result C=AxB is given by:

$$C = \begin{bmatrix} 0 & 6 & 12 \\ 0 & 6 & 12 \\ 0 & 6 & 12 \end{bmatrix}$$

Write an assembly program for matrix multiplication assuming all elements are integers and the matrices are stored in main memory. Initially, you may hard code the input matrices A and B of size 3x3 each in your program. You should store them in row-major order. Observe the cache performance (e.g., hit- rate, miss- rate etc.) for your code and try to increase the hit-rate. For this task you can use the Cache Simulator tool of the MARS software.

References and Resources

A. Software Download

[A.1] Java for Windows- Download and install the Java software for Windows through below link:
http://javadl.sun.com/webapps/download/AutoDL?BundleId=76860

[A.2] Java for Mac- Download and install the Java software for Mac through below link:
http://www.java.com/en/download/index.jsp

[A.3] MARS Software- Download MARS from the following link:
http://courses.missouristate.edu/kenvollmar/mars/download.htm

B. Tutorials

[B.1] P. Sanderson, and K. Vollmar, "An assembly language IDE to engage students of all levels: tutorial presentation," *Journal of Computing Sciences in Colleges* 22.4, pp. 122-122, 2007.

[B.2] Intro to MARS Video- You can watch the below tutorial video for basic information regarding the MARS software:
http://www.youtube.com/watch?v=z3ltaJ5UU5I

[B.3] Introduction to MIPS Assembly Programming- The below document by Oregon State University provides an introduction to MIPS Assembly programming in MARS:
http://web.engr.oregonstate.edu/~walkiner/cs271-wi13/slides/04-IntroAssembly.pdf

[B.4] MIPS Architecture and Assembly Language Overview:
http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm

[B.5] MIPS Memory Layout: Operating Systems I and Operating Systems and Process-Oriented Programming Courses at the Department of Information Technology, Uppsala University, Spring 2018.
http://www.it.uu.se/education/course/homepage/os/vt18/module-0/mips-and-mars/mips-memory-layout/

C. ASCII Table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com