Lab 2: System Calls, Branches, and Comparisons

Submission timestamps will be checked and enforced strictly by the CourseWeb; **late submissions will not be accepted**. Check the due date of this lab on the CourseWeb. Remember that, per the course syllabus, if you are not marked by your recitation instructor as having attended a recitation, your score will be cut in half.

In this lab, you are going to learn how to use various system calls (syscalls), how to compare numbers, and how to jump and branch.

Useful System Calls

There are a number of system calls integrated into MARS. For this lab, you are going to use system calls to print integers, print strings, read integer input from the keyboard, generate random numbers, and exit the program. To use a system call, you need to indicate which system call is to be executed by **setting the register `$v0` to the number associated with the desired operation**. Then, if needed, set arguments in the expected places (typically in the `$a` registers) and execute the `syscall` instruction. See "Help" → "Syscalls" for system call numbers, their associated arguments, and the registers in which those arguments should be placed.

Print Integer

To print an integer on the console screen, you need to set `$v0` to 1, set `$a0` to the value that you want to print, and then execute the `syscall` instruction. For example, to print the integer value stored in the register `$s0`, use the following code:

```
    addi $v0, $zero, 1          # Syscall 1: Print Integer
    addi $a0, $zero, $s0        # Set the value to print
    syscall                     # Print the integer
```

Print String

To print a string, you need to set `$v0` to 4, set `$a0` to the address of the first character of the null-terminated string to be printed, and then execute the `syscall` instruction. For example, the code below prints the string "Hello, world!" on the console screen:

```
.data
    helloMsg:   .asciiz    "Hello, World!"
.text
    addi $v0, $zero, 4          # Syscall 4: Print String
    la   $a0, helloMsg          # $a0 = address of the first character of helloMsg
    syscall                     # Print the string helloMsg
```

For a string to be printed to the console, the string must be null-terminated and it must be stored in the main memory. Note that the `.asciiz` directive in the `.data` segment above reserves a chunk of memory to store the null-terminated string "`Hello, World!`" and allows the memory address of its first character to be referred to by the label `helloMsg`. The instruction `la $a0, helloMsg` then puts the memory address referred by `helloMsg` into `$a0`.

Read Integer from Input

To read an integer from user input, you need to set `$v0` to 5 and execute the `syscall` instruction. When the syscall is executed, the program will wait for user to enter an integer on the console screen. Once user enters an integer and press Enter, the value entered by the user will be stored in the register `$v0` and control will return to the program. For example, the code fragment below sets register `$s2` to an integer provided by user input:

```
addi $v0, $zero, 5
syscall
add  $s0, $zero, $v0
```

System Time

The current system time in MARS is a count of the number of milliseconds since 00:00 UTC on January 1, 1970. This is a huge number which cannot be represented in a 32-bit quantity, so MARS uses two registers to store the current time as a 64-bit quantity.

To obtain the system time, set `$v0` to 30 and execute the `syscall` instruction. Once control is returned to the program, register `$a0` will contain the 32 least significant bits (or the "low-order bits") of the system time and `$a1` will contain the 32 most significant bits (or "higher-order" bits). For this lab, we will only use the low-order bits of the system time as a seed for our random number generator (RNG).

Set RNG Seed

As you may know, before generating random numbers, we need to set a seed for the random number generator (RNG). If we fail to do this, the RNG will generate the same set of random numbers every time the program is executed, which is usually not what we want. In MARS, you are allowed to have more than one random number generator but we will use only one for this lab.

To set the seed for an RNG, you need to set `$v0` to 40, set `$a0` to an ID number identifying the RNG (this can be any number or simply 0), set `$a1` to a seed value (typically the low-order bits of the current time), and then execute the `syscall` instruction. **Note** that you need to set seed only once for each RNG.

Random Integer

After an RNG seed is set, we can generate a random number between 0 (inclusive) and an arbitrary upper bound (exclusive) using the random integer range system call.

To generate a random integer between 0 (inclusive) and n (exclusive), set `$v0` to 42, set `$a0` to the RNG ID you used when setting the RNG seed, set `$a1` to the desired value of n, and execute the `syscall` instruction. When control is returned to the program, the randomly generated number will be stored in register `$a0`. The code below shows how to generate a random number between 0 (inclusive) and 5 (exclusive) and store the result in register `$s1`, assuming that the RNG ID is 0:

```
addi $v0, $zero, 42          # Syscall 42: Random int range
add  $a0, $zero, $zero       # Set RNG ID to 0
```

```
    addi $a1, $zero, 5          # Set upper bound to 5 (exclusive)
    syscall                     # Generate a random number and put it in $a0
    add  $s1, $zero, $a0        # Copy the random number to $s1
```

Terminate Program

It is always a good idea to explicitly terminate your program, rather than letting the executing "fall off" the end of the list of instructions. To terminate a program, simply set `$v0` to 10 and execute the `syscall` instruction.

Branches and Comparison

In MIPS assembly language, you are allowed to move to an arbitrary point in a program, either when a condition is satisfied (a "branch") or without any condition at all (a "jump"). A label is used to indicate such a point in a program, and it can be located anywhere in your program, whether before or after the branch/jump instruction.

There are two main branch instructions in MIPS, `beq` (branch if equal) and `bne` (branch if not equal). **Both of these instructions take two registers and one label as their operands**. For example, when the instruction `beq $s0, $t5, foo` is executed, it will make the program jumps to the location indicated by the label named `foo` if the values stored in register `$s0` and `$t5` are equal. Otherwise, the program will continue to the next instruction immediately following the `beq` instruction.

Likewise, when the instruction `bne $s3, $zero, bar` is executed, it will make the program jump to the location indicated by the label named `bar` if the value stored in the register `$s3` is **not** equal to 0. Otherwise, it will continue to the next instruction.

Hopefully, you can begin to see how we might be able to accomplish some interesting control flow patterns with these instructions, similar to what `if`, `else`, and `else-if` blocks might allow us to do in a higher-level programming language.

For an unconditional jump, use the instruction `j`. Since there is no condition to be met, `j` **only needs a label as its only operand**. Combined with branches, this is a useful way to implement looping constructs like `while`, `do-while`, and `for` would offer in higher-level languages. For example, `j loop` will make the program jumps to the location indicated by the label named `loop`.

The code below shows how to use these instruction to "countdown" from 5 to 1, while skipping 3:

```
    addi $s0, $zero, 3          # Set $s0 to 3 (do not want to print 3)
    addi $s1, $zero, 5          # Counter (start at 5)
loop:
    # If integer to be printed is 0, we are done (so go to done)
    beq  $s1, $zero, done
    # If integer to be printed is not equal to $s0 (3), go to printInteger
    bne  $s1, $s0, printInteger
    addi $s1, $s1, -1           # Decrease the counter by 1
```

```
    j    loop                   # Go back to loop
printInteger:
    addi $v0, $zero, 1          # Syscall 1: print integer
    add  $a0, $zero, $s1        # Integer to be printer (from $s1)
    syscall                     # Print integer
    addi $s1, $s1, -1           # Decrease the counter by 1
    j    loop                   # Go back to loop
done:
    addi $v0, $zero, 10         # Syscall 10: terminate program
    syscall                     # Exit
```

To bring this all together, we need a good way to compare the values stored in two registers. The most commonly used instruction for this purpose is `slt` (set if less than). For example, `slt $t0, $s1, $s3` will set the value of the register `$t0` to 1 if the value stored in register `$s1` is less than the value stored in the register `$s3`. Otherwise, the register `$t0` will be set to 0. We generally use `slt` together with branch instructions (`beq` and `bne`) and jump (`j` to achieve behavior similar to what `if`, `else`, and `else-if` blocks would provide in higher-level languages.

Higher/Lower Game

Your task is to write a program that generates a random number between 0 and 9 (inclusive), then ask user to guess it. If the user gets it right, congratulate them. They have won and the game ends. Otherwise, you should tell them whether their number is too high or too low and let them guess again.

The user is allowed to guess at most three times. If the user cannot get it right after the third time, they have lost and they game ends. Make sure to prompt the user with a message and explain whether their guess is correct, too low, too high, etc. The examples below show a couple of game plays:

```
Enter a number between 0 and 9: 4
Your guess is too low.
Enter a number between 0 and 9: 7
your guess is too low.
Enter a number between 0 and 9: 8
your guess is too low.
You lose. The number was 9.
-- program is finished running --

Enter a number between 0 and 9: 3
Your guess is too low.
Enter a number between 0 and 9: 7
your guess is too high.
Enter a number between 0 and 9: 4
Congratulation! You win!
-- program is finished running --
```

Create a "higher/lower" game program as described above. Please exactly match the format of the messages provided. Save your program as `lab02.asm`.

Submission

Submit your `lab02.asm` file via CourseWeb before the due date stated on the CourseWeb.