# Week 8 Lab

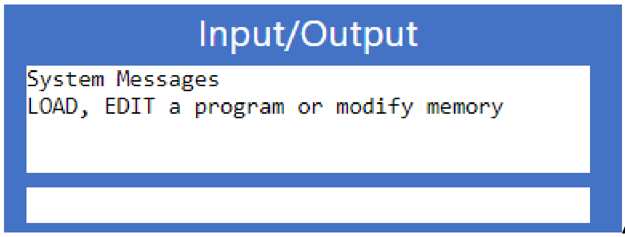
## Matchsticks

In this lab you will be writing a very simple game called Matchsticks. The game starts with a pile of 15 matchsticks (although it could be any number). Players take it in turns to remove either 1,2, or 3 matchsticks from the remaining pile. A player wins the game by forcing the opponent to take the last matchstick. Our implementation will pit a single human player against the computer.

To implement the game, you will need to implement iteration (looping), and selection (branching), in ARM assembly language. You will also need to learn some patterns for writing text to the console, and reading inputs from the user during the game. All of this requires first being comfortable with the concept of loading and saving of values from/to memory locations using the LDR – 'LoaD Register (from memory)', and STR – 'STore Register (to memory)', instructions.

## Simple Input/Output

The central region of the ARMlite GUI includes an area labelled Input/Output. The topmost field is the console, which offers a way of displaying text to the user. The smaller second field below this is an area where input data can be obtained from the user upon request.



```
System Messages
LOAD, EDIT a program or modify memory
```

As was shown in the lecture videos, w riting to and reading from these fields involves the use of instructions STR and LDR, along with labels representing specific addresses. We're going to apply these ideas now in the content of the Matchsticks game we are developing.

### *Excercise 8.1.1*

We're going to build our game up iteratively, and to start with, consider only 1 player. Lets start with some basic output to tell the player how many matchsticks are left (initially 15). We want the output to read "15 remaining" In ARMlite, write the code needed to do this.

If you need some further guidance, consider implementing this in the following steps):

1. create a label for the ASCII text (i.e., type .ASCIZ) you want to display (i.e, " remaining")
2. initialise register R0 to 15
3. write the value inside R0 to the output display
4. write the string " remaining"

### *Excercise 8.1.2*

Now lets consider getting some input from the user. Recall each player needs to provide a number between 1 and 3 (ie., the number of match sticks to remove). Building on what you implemented in 8.1.1, write the assembly code required to prompt the user for input using the string "How many do you want to remove (1-3)?", and then read in an integer value and store it in a register of your choice (but not one already being used!)

*Hint - as in 8.1.1, define  a label to store the string you want to display, and recall from this week's video lectures how numbers can be read in using LDR*

### *Excercise 8.1.3*

So now we have the player's input: the number of matchsticks to remove.  We now need to calculate how many matchsticks remain once this number is removed.  Again building on the code you wrote in the previous two exercises, write the code required to calculate the remaining number of matchsticks, and store this number into R0 (which recall, you initialised to 15 in 8.1.1).

***Once complete, show your tutor the executing program, and take a screen shot showing the code and output, and include in your submission document.***

## Branching and Looping

Now things start to get interesting.  We'll continue, for now, assuming there is only one player.  As such, we will want to develop a program that repeatedly asks the player to enter a number of matchsticks to remove, until the total number of matchsticks remaining is 0.  Whenever your program needs to repeatedly do something, this almost always means you need to implement branching and looping, which we discussed in this week's video lectures.  *Before you proceed, we assume you have watched those videos!*

Lets start with the simplest way to loop, using the "**B**" instruction (which refers to "**B**ranch".

### *Excercise 8.2.1*

Building on the code you developed in Part 8.1 of this lab, add the code required to make the program repeatedly:

1. display how many matchsticks are remaining;
2. prompts the user for a number between 1-3;
3. reads the value entered; and,
4. calculates the remaining amount

In total, this exercise should only require you to add/modify two lines of code

*Hint:  you will need a label to branch back to.  Think carefully about where this needs to go!*

Once complete, run your program and verify that the displayed number of matchsticks remaining  decreases in accordance with the number entered each iteration.

***What happens if you enter a number that takes the number of matchsticks remaining beyond 0 (i.e., into negative values) ?  What do you think is going on here ?  Hint - take a look at the value in the register!***

**Show your program  and discuss your answer with your tutor, and take a screenshot of your current program and output and include in your Submission Document.**

### *Excercise 8.2.2*

The program so far does no validation of the user provided input, which is supposed to be a number between 1 and 3.  A way to handle this is to repeatedly ask for the input to be entered until the user provides a valid number.  This requires another loop to be implemented inside the loop you just created, however this time the loop only happens under a certain condition, or more precisely a comparison of two values.

**Question 8.2.2(a) - What is the condition that needs to be satisifed in order for this loop to occur ? Write this as a comparison using an inequality (ie., less than, greater than, less than or equal, greater than or equal)**

**Question 8.2.2(b) - What two ARM assembly instructions could be used to create a branch that only occurs under this condition ?**

Recall from the video lectures that a any operation on values value comparison in ARM assembly causes an update of the so called Application Program Control Register (APCR). Specifically, the 4 status bits **N**egative**, Z**ero, **C**arry and o**V**erflow are updated to reflect the outcome of an operation.

Of most relevance to value comparison are:

- **N,** which is set if the second value (V2) being compared is larger that the first value (V1) being compared (ie., CMP V1 V2)
- **Z,** which is set if the two values are equal

The use of this bits when the CMP instruction is called reflects the fact that CMP is really performing substraction of the two values being compared (i.e., CMP V1 V2 ==> V2 - V1.

This allows the comparison to occur in a single CPU cycle, making it very fast.

**Question 8.2.2(c) - Based on the instructions you outlined in 8.2.2(b), what status bit would be set to 1 if the loop was to repeat ?**

Now lets make the changes we need to make to the program.

**Question 8.2.2(d) - What are all the modifications needed to the current program to implement this feature ? Make the required modifications to your program to perform the task.**

***Enter your answers and a screen shot of your current program and test output into your Submission Document.***

# Implementing a Computer Player

We've now got a program that allows a player to keep removing matchsticks until none are left. We now want to implement a computer player to play against.

The computer player will not be very sophisticated. We will implement it to select a random number of matchsticks to remove between 1 and 3.

In ARMlite, a random 32 bit pattern can be generated and placed in a register using the following instruction:

```
LDR R2, .Random
```

This above code loads Register R2 with the random 32 bit pattern.

### *Excercise 8.3.1*

For our purposes, we only need a random number between 1 and 3, which means we only care about the 2 least signficant bits in the randomly generated 32-bit pattern.

**Question 8.3.1(a) What bit-wise operation can we perform on the register holding the 32 bit pattern to set all bits in the register to zero except the least signficant 2 bits ? Write this as a single line of code.**

If your answer to the above question is correct, then if executed, the register holding the random number would a value between 0 (i.e, 0x00000000) and 3 (0x0000000F)). This is close to job done, except "0" is not in the range 1-3. To handle this, we can check if the resulting value is 0, and if it is, ask for a new random number. We can repeat this until the condition is no longer satsified (i.e., the random number is not 0.

So let's write a program to do this.

**Question 8.3.1(b) Using a label named "`select:`" Write the code needed to repeatedly sample a random number (from .Random) until the value is in the range 1-3. For now, just write this as a seperate program and test it.**

You can inspect the contents of the register to verify the code is working correctly.

***Enter your answers and a screen shot of your current program and test output into your Submission Document.***

## *Excercise 8.3.2*

Now let's consider how we integrate the above code into our matchsticks program. So far the code we wrote above repeatedly requests a random number (i.e, the number of matchsticks selected by the computer player to remove) until the number is between 1 and 3. To integrate with our game, we also need to check that the selected value does not exceed the number of matchsticks left to remove.

Consider the matchsticks program you have written so far, and identify which register holds the remaining number of matchsticks - *let's assume its R0*.

Also, before you copy in any code from **Question 8.3.1,** make sure the register you choose to read random numbers into is not a register already being used in your matchsticks program (this is very important!) - *let's assume it's R2*.

We now want to expand our random number selector so that it repeatedly asks for a random number until the value obtained is valid (i.e, between 1 and 3), and also doesn't exceed the number of matchsticks remaining.

The psuedo code for this might be something like:

```
select:  read random number into R2

         set all bits to zero except least signficant 2 bits

         Compare R2 with value 0

         Branch to select if R2 == 0

         Compare R2 with R0

         Branch to select if R0 < R2
```

**Question 8.3.2(a) - Write the ARM assembly code that implements the algorithm expressed in the psuedo code above. Implement this as a seperate stand alone program and initialise R0 with a**

**number at the start of your program to allow you to test the functionality. You wil want to test it using different values in R0.**

***Enter your answers and a screen shot of your current program and test output into your Submission Document.***

## Implementing the full game (not required to be complete to get "the lab mark")

It's now time to integrate your computer player code into the matchstick game code, and add the code to determine the winner. For this we're taking the training wheels off, and just letting you have a go. ANy reasonable attempt will get you the mark, so don't stress if you cannot fully complete this. However, have a good go and see how far you can get. Ask your tutor for help and tips if you need to.

### *Excercise 8.4.1*

Using the code you developed in the Part 8.3, open up your matchsticks program and implement your program so that the computer, and then the human player, take turns in removing 1-3 match sticks. You will need to think carefully about where the computer player code belongs. Also consider including some extra messages to output to the user to let them know when the computer has had a turn.

### *Excercise 8.4.2*

Finally, add the code you need to determine who the winner is. Remember that a player wins by forcing their opponent to take the last matchstick.

***Copy the code of your solution (or reasonable attempt) into your Solution Document.***

*If you have something up and going - great job ! Show you tutor in class.*

*If you didn't quite get there - no worries ! Have a chat with your tutor and get some tips and advice. You should continue to try and get it going. It's a great way to study and prepare for the assignment.*