# Preparation

1. Set the simulation into Def Fast mode
2. Use a text editor to create and edit the code file, saving it locally each time, before loading it into the simulation to run. (This is safer than editing the code directly in the simulator).

# Story 1: Draw the starting snake and apple

## Requirement

Create a two-segment snake (head + tail) near the middle of the screen. The snake should be a green colour. Add an apple (a single pixel of a different colour) somewhere below the snake.

## Techniques

The simulation provides ‘addressable video memory’, that runs from (word) location 256 (0x100 in hex) for the top-left corner, to 1023 (0x3ff in hex) for the bottom-right.

Exercise

What does the following code produce?

mov r0,#0

str r0,256

str r0,1023

**[Peter: assembler does not accept direct addressing to the screen memory, I think it should]**

[Paste in a partial screenshot showing only the output window]

mov r0,#1, means ‘move into register 0, the immediate value 0’. This is known as ‘immediate address mode. In this case, the value 0 represents the colour black (no colour).

When the program starts register 0 should default to the value 0, but it is not safe to assume this, so we set it to 0 explicitly. This is equivalent to the practice of initialising all variables in a higher level language.

We can specify other colours using the same RGB (Red Green Blue) format as used when creating a web page. This is best specified in hex, so for example, 0x008844 results in a suitable hue of green for the snake.

## Implementation

Create and save a new file called Snake, and add the following code:

defineRegisters:

mov r12,#0x008844 //Snake colour (green)

drawSnake:

str r12,527 //Tail

str r12,528 //Head

Load the file into the simulator, assemble and run.

Paste in a partial screenshot showing the assembled code and the Output after running.

Notice that we have added three labels: defineRegisters:, drawSnake:, and drawApple:. These aren’t actually used by the program at this point, but they make the code more readable. [Paste in a partial screenshot showing the assembled code and the Output after running].

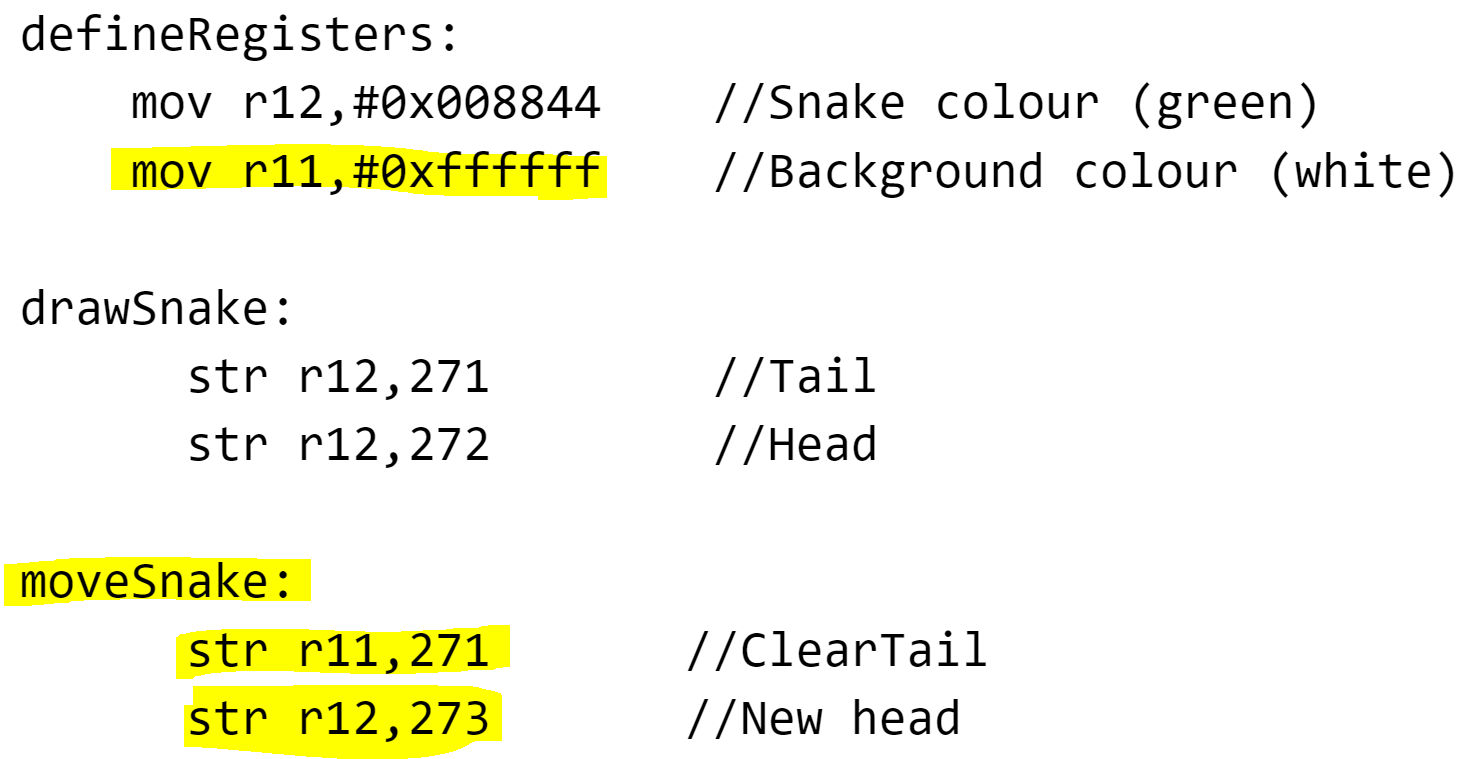
Why have we switched from using r0 to r12 for the green colour? We are simply adopting a common convention to use the lower-number registers for handling variable pieces of data and the higher-number ones for constants.

# Story 2: Move the snake

## Requirements

## Techniques

To move the snake one pixel to the right we *could* draw a new head in the next screen memory location (273) and then reset the tail (271) to the background colour (white), as shown below (*don’t modify your code yet):*



The problem with this approach is that it won’t *generalise*. We will have to add two new instructions for each pixel that the snake moves, and we won’t be able to vary it (eventually) based on live instructions from the player.

So we are going to *refactor* the code from story 1, making use of two more registers to hold the position of the head and tail:

mov r3, #271 //Head position, initialised  
 mov r4, #272 //Tail position, initialised

And then we are going to use these registers in the drawSnake routine, using *indirect addressing mode*, signalled by square brackets:

drawSnake:  
 str r12,[r4] //Tail  
 str r12,[r3] //Head

The first line can be read as ‘store the value held in r12 (the snake colour) into the memory address that is held in r4 (i.e. initially, memory location 272). *Indirection* lies at the heart of many advanced programming techniques.

Having done this, we can adjust the values held in r3 and r4 to point to new locations and then use the same store instructions to re-draw it. And if we do this in a loop then we can move the snake continuously to the right.

## Implementation

Modify your code to look like this.

defineRegisters:

mov r12,#0x008844 //Snake colour

mov r11,#0xffffff //Background colour (white)

mov r3, #528 //Head position, initialised

mov r4, #527 //Tail position, initialised

drawSnake:

str r12,[r4] //Tail

str r12,[r3] //Head

moveSnake:

str r11,[r4] //Reset tail to Background

add r4,r4,#1 //Increment the tail pointer

add r3,r3,#1 //Increment the head pointer

str r12,[r3] //Draw new head

b moveSnake //Loop

Modify your code from Story 1 (all new lines, and changes to existing lines, are highlighted).

Copy the code into the simulation, assemble, and run.

What happens when the snake gets to the right hand edge of the screen area, and why?

If you leave the program to run long enough you will get an error.

On which instruction number has the error occurred?

Which register is being used in that instruction, and what value is it holding at that time?

Why does this cause an error?

# Story 3: Add an apple, and allow the snake to eat it

## Requirements

At the start, draw an apple (one pixel of a different colour) in a position below the starting point of the snake such that the snake will pass over it. When this happens, the apple should disappear. (In a later story we will want to position the apple randomly.

## Implementation

We have already learned the patterns we need, so make the changes highlighted below.

defineRegisters:

mov r12,#0x008844 //Snake colour (green)

mov r11,#0xffffff //Background colour (white)

mov r10,#0xff8800 //Apple colour

mov r3, #528 //Head position, initialised

mov r4, #527 //Tail position, initialised

mov r5, #750 //Apple position

drawApple:

str r10,[r5]

drawSnake:

str r12,[r4] //Tail

str r12,[r3] //Head

loop:

str r11,[r4] //Reset tail to Background

add r4,r4,#1 //Increment the tail pointer

add r3,r3,#1 //Increment the head pointer

str r12,[r3] //Draw new head

b loop

# Story 4: When the snake eats the apple, make it grow

## Techniques

We can detect the event of ‘eating’ the apple, by checking, within the loop, when the snake’s head position matches that of the apple. We can the grow the snake length by one, simply by not updating the position of the tail for that cycle of the loop.