# 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) a little 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 - in a higher level language - of initialising all variables.

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 r1,#0x008844 //Snake colour (green)

drawSnake:

str r1,527 //Tail

str r1,528 //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 used r1 rather than r0? It is a common convention, though my no means necessary, to keep r0 for temporary purposes.

# 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 (529) and then reset the tail (527) to the background colour (white) something like this (don’t make these changes to your code):

defineRegisters:

mov r1,#0x008844 //Snake colour

mov r2,#0xffffff //Background colour (white)

drawSnake:

str r1,527 //Tail

str r1,528 //Head

moveSnake:

str r2,527 //Blank out the tail

str r1,529 //Add head in new position

The problem with this approach is that it doesn’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 directions from the player.

So, instead, 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, #528 //Tail position, initialised

mov r4, #527 //Head 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,[r3] //Tail  
 str r12,[r4] //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 527). *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 r1,#0x008844 //Snake colour

mov r2,#0xffffff //Background colour (white)

mov r3, #528 //Tail position, initialised

mov r4, #527 //Head position, initialised

drawSnake:

str r1,[r3] //Tail

str r2,[r4] //Head

moveSnake:

str r2,[r3] //Reset tail to Background

add r3,r3,#1 //Increment the tail pointer by 1

add r4,r4,#1 //Increment the head pointer by 1

str r1,[r4] //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: Refactor to use indexed addressing

## Requirement

This story does not add any value to the user. It is an example of *refactoring* – which means ‘improving the design of your code without changing functionality’. The motivation for refactoring may be to make it easier to change/extend the design in future (in response to new requirements), to make it more efficient in execution, or simply to make the code easier to read/understand (which will in turn make it easier to change in future).

## Techniques

We are going to make use of another new addressing mode known as *indexed addressing* whereby the address is calculated from a base address plus a variable amount (the index value) that is held in a register.

## Implementation

We learned in Story 1 that screen memory runs from locations 256 to 1023. So far we have set the position of the snake *absolutely* e.g. using 528 as the starting position of the head, somewhere near the middle of the screen.

It would be more elegant, and make position-related calculations easier, if we could make all such positions *relative* to the start of screen memory. Thus, location 528 would be replaced by 272 (528 = 256 + 272). The assembler allows us to do this using *indexed* addressing mode, as shown below.

defineRegisters:

mov r1,#0x008844 //Snake colour

mov r2,#0xffffff //Background colour (white)

mov r3, #271 //Tail position, initialised

mov r4, #272 //Head position, initialised

drawSnake:

str r1,[r3+256] //Tail

str r1,[r4+256] //Head

moveSnake:

str r2,[r3+256] //Reset tail to Background

add r3,r3,#1 //Increment the tail pointer by 1

add r4,r4,#1 //Increment the head pointer by 1

str r1,[r4+256] //Draw new head

b moveSnake //Loop

Each of the changed versions above can be read as ‘calculate the address based on 256 plus the value held in the register’. In each case the *base* address is fixed at 256 and the *index* (added to that base address) is provided by the specified register (r3 or r4).

# Story 4: Add an apple, and allow the snake to eat it

## Requirements

At the start of the game, draw an apple (one pixel of a different colour) in a position a little below the starting point of the snake. (In a later story we will want to position the apple randomly.) When the snake will passes over the apple, the apple should disappear.

## Implementation

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

defineRegisters:

mov r1,#0x008844 //Snake colour

mov r12,#0xffffff //Background colour (white)

mov r3, #528 //Head position, initialised

mov r4, #527 //Tail position, initialised

mov r5, #330 //Apple position

mov r6,#0xff8800 //Apple colour

drawApple:

str r6,[r5+256]

drawSnake: (unchanged)

moveSnake: (unchanged)

When you run the game, and the snake has passed over the apple, what happens to the apple, and why?

Paste in a screenshots showing the snake just about to hit the apple and just after passing over it completely. (Hint: you can hit Stop on the simulator at any point and then hit Run again to resume execution).

# Story 5: Grow the snake

## Requirement

When the snake ‘eats’ the apple, the snake’s length should grow by one.

## 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.

## Implementation

defineRegisters: (unchanged)

drawApple: (unchanged)

drawSnake: (unchanged)

moveSnake:

add r4,r4,#1 //Increment the head location by 1

cmp r4,r5 //If the head is in same location as apple...

beq moveHead //...Skip updating the tail, to make snake grow

str r2,[r3+256] //Reset tail to Background

add r3,r3,#1 //Increment the tail pointer by 1

str r1,[r4+256] //Draw new head

b moveSnake //Loop

Having made the modifications and tested that it works, *temporarily*  add code to draw two further apples lower down the screen. Test that the snake grows with each apple eaten, and paste a screenshot of the Output area showing the snake having grown to 5 segments.

# Story 6: Change of direction

## Requirements

When the S key is pressed, switch to moving downwards. This snake should continue downwards until its head reaches the bottom of the screen.

## Techniques

The following code propably looks like it *should* work, but it actually contains a subtle bug.

defineRegisters: (unchanged)

drawApple: (unchanged)

drawSnake: (unchanged)

moveSnake:

inp r0,4

cmp r0,#83 //S key

beq down

right:

mov r0,#1 //Re-purpose r0 to hold an increment of 1

b reDraw

down:

mov r0,#32 //32 moves down one row on screen

reDraw:

add r4,r4,r0 //Increment the head pointer by value of r0

cmp r4,r5 //If the head is in same location as apple...

beq moveHead //...Skip updating the tail, to make snake grow

str r2,[r3+256] //Reset tail to Background

add r3,r3,r0 //Increment the tail pointer by value of r0

str r1,[r4+256] //Draw new head

b moveSnake //Loop

**What happens when you run the game and hit the ‘s’ key to change direction, downwards?**

**Try stepping through the program, hitting the ‘s’ key early to see if you can see what the bug is?**

The bug is quite subtle. Upon the change in direction we immediately start moving both the head and the tail pointers (r3 and r4) downwards. But because the tail is (initially) one to the left of the head, they are going to move down next to each other, not in the same column of pixels, so the new trail created by the advancing head, is never reset to the background colour by the advancing tail. And if you change direction after the snake has eaten the apple and grown to three segments, the head and tail will move downwards further apart.

[add diagram]

To fix this properly, and, especially, to cope with later versions of the game where the snake may acquire a complex shape from many turns, we really need to ensure that the tail always follows the same path as the head, lagging behind by as many segments as the snake is long.

Could this be done by getting the tail-update routine to read the screen memory, looking to see in which direction is the next snake-coloured pixel. That might work for simple cases, but won’t work if the snake has doubled-back on itself e.g.

[add diagram]

The proper solution is to keep a record of locations the head has passed through, elsewhere in memory. So, before we can continue, we need to do another refactoring.

Before moving onto the refactoring, undo the changes made in this story i.e. revert to the code at the end of story 5.

Being willing to ‘back track’ in your coding is an essential part of good coding practice. Don’t be tempted to hold onto the wrong code just because you spent time writing it!

# Story 7: Refactoring

## Requirement

Without changing the functioning of the game (from Story 5), change the implementation such that it maintains a record of the positing of the snake’s head in a data structure within main memory.

## Techniques

We can declare a data structure called ‘body’ with that label and the pseudo-instruction dat, at the end of the program code:

body: dat 0

The 0 here just sets the value held in the first memory location of body to a default value zero (because the dat pseudo instruction requires a value).Unless we need another data structure, this one can extend from the address of that label to the end of the memory (location 199).

The actual data structure we will use is somewhat like the ‘queue’, which you should be familiar with from studying Data Structures. We will have two pointers, pointing to the front and back of the data in the queue, corresponding to the position of the tail and head respectively. That might sound the wrong way round, but think about it: each time the head moves we will *enqueue* a new value (the head’s new location) i.e. add it to the back of the queue and change the pointer; and each time the tail moves we will *dequeue* (and discard) the value at the front of the queue, updating the front pointer.

So, even without growing, the data holding the locations (in video memory) of the current segments will move down the data area, until the limit of memory is reached. To avoid this wrecking the game we need to make our implementation a *circular queue*, such that when either pointer reaches the end of memory, it is reset to the start of the body data area. This can continue until the snake becomes as long as the available memory, in which case the game must be terminated – the player has achieved the maximum permissible length of snake!

[diagram needed here to show operation of queue. Also note that its operation can be observed directly in the memory area of the simulator].

## Implementation

defineRegisters:

mov r1,#0x008844 //Snake colour

mov r2,#0xffffff //Background colour (white)

mov r3, #527 //Tail position, initialised

mov r4, #528 //Head position, initialised

mov r5, #330 //Apple position

mov r6,#0xff8800 //Apple colour

mov r7, #body //Pointer front of queue, initialised to first data loc

add r8,r4,#1 //Pointer to head address in body data (1 after tail)

InitialisePointers:

str r3, [r7] //r4 points to the tail address

str r4, [r8] //r3 points to the head address

drawSnake: (unchanged)

drawApple: (unchanged)

moveSnake: (unchanged)

moveTail:

ldr r0, [r7]

str r2,[r0+256] //Reset tail to Background

add r7,r7,#1 //Increment the tail pointer (for use next cycle)

moveHead:

add r8,r8,#1 //Increment the head pointer

str r4, [r8] //Store the new head location in data

str r1,[r4+256] //Draw new head

b moveSnake

body: dat 0 //body segment pointers extend from here to end of memory (addr 199)

# Story 6c: When data area if full, loop the pointers around to the start

TODO: Keep this as story 8. Convert queue to a circular queue.

## Techniques

Implement this like a circular queue that uses an array with two pointers (except that here we only ever add to the queue).

moveTail: (no changes before this)

ldr r0, [r7]

str r2,[r0+256] //Reset tail to Background

add r7,r7,#1 //Increment the tail pointer (for use next cycle)

cmp r7,#200 //Check pointer is still within memory

blt moveHead

mov r7, #body //If not loop pointer back to start of body data

moveHead:

add r8,r8,#1 //Increment the head pointer

cmp r8,#200 //Check pointer is still within memory

blt updateData

mov r8, #body //If not loop pointer back to start of body data

updateData:

str r4, [r8] //Store the new head location in data

str r1,[r4+256] //Draw new head

b moveSnake

body: dat 0 //body segment pointers extend from here to end of memory (addr 199)