Week 9 Lab

Indirect and Indexed Addressing

Before this week we had seen how LDR and STR can be used to access specific memory locations we have defined as either a number (in hex) or as a label which is then translated into a number by the assembler. We then added to this the concept of *indirect addressing*, in which addresses may be stored in a register and passed to STR or LDR. Beyond that, we then saw how offsets can also be provided to refer to specific words/bytes beyond the address specified in the register - we called this *indexed addressing*.

In this lab we are going to explore both indirect and indexed memory addressing in the context of ARMlite's graphics display. ARMlite has three modes of graphics display: Low-res (32x24 pixels), Medium-res (64 x 68 pixels), and High-res (128 x 96 pixels). Pixels are defined as 24 bit values, with 8 bits dedicated to each of Red, Green and Blue (ie., each colour component has 256 possible intensities, with 0 being darkest (ie black), and 255 being brightest).

We talked about Low-res in Week 8. In Low-res Mode, direct addressing is used to write specific 24 bit (3 byte) RGB pixel colour values to a specific location. That is, every pixel in the display has an associated label, following the naming convention `.Pixel0` all the way to `.Pixel767`. To warm up, lets play with the Low-res graphics mode.

Exercise 9.1.1

Consider the following code, which draws a single red horizontal line of length 20 pixels in the top row of the display in Low-res display mode:

```
MOV R1, #.red
STR R1, .Pixel0
STR R1, .Pixel1
STR R1, .Pixel2
STR R1, .Pixel3
STR R1, .Pixel4
STR R1, .Pixel5
STR R1, .Pixel6
STR R1, .Pixel7
STR R1, .Pixel8
STR R1, .Pixel9
STR R1, .Pixel10
STR R1, .Pixel11
STR R1, .Pixel12
STR R1, .Pixel13
STR R1, .Pixel14
STR R1, .Pixel15
STR R1, .Pixel16
STR R1, .Pixel17
STR R1, .Pixel18
STR R1, .Pixel19

HALT
```

(a) Write a simple ARMlite assembly program that draws a single line of the same length across the second row (starting from the left-most column) in Low-res display mode.

(b) Add to your assembly program code that draws a single line of the same length vertically, down the middle of the display in Low-res display mode

Once complete, show your lab demonstrator, and take a snap shot of your screen and include in your lab submission document

In contrast to Low-res, the Mid-res (and High-res) display mode has no labels specifically referring to each pixel address. Rather, you need to need use indirect or indexed addressing to access pixel bytes. That is, thinking of the display memory as a long array of 4 byte words, where each word is holding the 24 bit colour value for that pixel, starting from top-left (with memory address #.PixelScreen) and extending row-wise to the bottom-right pixel, you can access any pixel by calculating the number of bytes from the start (ie pixel position number in display x 4 bytes). Let's have a play in Medium-res mode.

Exercise 9.1.2

Consider the following code in Medium-res display mode, which like the code above, draws a straight red line of length 20 pixels.

```
MOV R1, #.PixelScreen   // base address of the medium and high res pixel display memory
 MOV R2, #.red
 MOV R3, #0
loop:
 ADD R4, R1, R3     //  calculate the byte offset (R1 + R3) for the next pixel and store new address
in R4
 STR R2, [R4]
 ADD R3,R3,#4
 CMP R3, #80
 BLT loop
HALT
```

This is an example of using indirect addresssing. Step through the code yourself and make sure you understand how its working. You may want to verify your understanding with your lab demonstrator before you move on.

Exercise 9.1.3

(a) Explain what specifically makes this code an example of indirect addressing ? How is it using indrect addressing to draw each pixel ?

(b) Once you're confident yo understand the code, modify the program so that it draws a line of the same length along the second row of the Mid-res display.

Hint - think about how many bytes the start and end of this line must be from the #.PixelScreen (the top-left corner pixel of the display)

(c) Further modify your program so that it also draws a line of the same length vertically down the middle of the display.

Hint - in this case the pixels to draw are not adjacenet to each other in memory, but they are a constant number of bytes apart from each other. Think about how many bytes you need to jump to get to the same position in the next row.)

Once complete, show your lab demonstrator, and take a snap shot of your screen and include in your lab submission document

Indexed Addressing

In lectures we also discussed indexed addressing, in which we can use a base address which is kept constant, and a byte offset to access specific locations in memoery beyond the base address. Lets have a play with indexed addressing with our medium-res graphics.

Excercise 9.2.1

Consider again the code below which we looked at in the last part:

```
 MOV R1, #.PixelScreen
 MOV R2, #.red
 MOV R3, #0
loop: ADD R4, R1, R3
 STR R2, [R4]
 ADD R3,R3,#4
 CMP R3, #80
 BLT loop
HALT
```

In the example above, the indirect address, held in R4, is made up of from a constant value held in R1, (the starting address for the grid of pixels, #.PixelScreen) plus a variable number of bytes (in R3). We could say that the value in R1 is the 'base' address and the value in R3 is a variable 'index', added to the base. This suggests the use of '*indexed addressing*', which as discussed in this week's lectures, is often used when accessing individual locations within a pre-defined blocks of memory.

In ARMlite, rewrite the code above so that is uses *indexed addressing* to draw the line in Medium-res display mode.

Copy and past a screenshot of your code and correct output into your submission document.

Excercise 9.2.2

Let's now extend your solution in 9.2.1 to produce 10 consecutive rows containing a line of length 80 pixels. For this you must use indexed addressing (using .PixelScreen as the base address, and should not duplicate any code to draw each line on the display (ie., use loops).

When complete, *copy and past a screenshot of your code and correct output into your submission document.*

Arrays

Arrays are a fundemental data structure in computer systems, for which almost all programming languages support in one way or another. An array is simply a contiguous, indexable block of memory for storing values of the same size (i.e., bytes). From an assembly programming perspective, an array is simply a labelled memory address indicating the beginning of a block of pre-allocated memory. Each cell within can then be indexed as an offset number of b bytes from the start address. This workds because every cell is exactly the same size, and so as long as you know how many bytes per "cell", then you can use indexed addressing to store and load values to and from any cell in the array.

So let's play with arrays.

Before you begin, enter the following code into ARMlite:

```
.ALIGN 256
arrayLength: 10
arrayData: 9
 8
 7
 6
 5
 4
 3
 2
 1
 0
```

Excercise 9.3.1 (a)

The above code defines an array of 10 32 bit integers. What is the purpose of the `.Align 256` instruction ?

Excercise 9.3.1 (b)

Add a line of code to the above to read the 5th value of the array to register R0 (i.e,. it should use indirect addressing to access the 5th cell in the array)

Excercise 9.3.1 (c)

Now modify your code so that the index to read from in the array is provided in R1.

Take a screen shot showing the code and output, and include in your submission document.

Excercise 9.3.2

Now modify your code so that it adds up all the values in the array. Your program should use indexed addressing to access each value and write the result to R0.

Take a screen shot showing the code and output, and include in your submission document.

Excercise 9.3.3

Using the original array definition, modify your code so that it adds up all the values in the array. Your program should use indexed addressing to access each value and write the result to R0.

Take a screen shot showing the code and output, and include in your submission document.

More Arrays

Excercise 9.4.1

Using the original array defintiion give in Part 9.3, write an ARMlite program copies all the values from this array into another array of equal size (in reverse order).

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

Excercise 9.4.2

Using the original array definition, write an ARMlite program that reverses the order of the values in the array (*without using another array*)

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