CS061 - Lab 04
Arrays

1 High Level Description

Today's lab will examine the difference between numbers and digits *(huh??)*, and further explore the mysteries of console output.

2 Our Targets for This Week

0. Lab 03 review – Exercise 01
1. What a character! Exercise 02
2. Bits and pieces – Exercises 03 - 05
3. I rock!

3 Ok, let's obliterate these targets!

3.1 Lab 03 review

You will be building on the skills you learned last week: prompts, sentinel-controlled loops, and arrays, so let's start with a variation on last week's array – this time storing the results of a calculation, rather than user input.

Exercise 01
Use .BLKW to set up an array of 10 values, starting at memory location x4000, as in lab 3.
Now *programmatically* populate the array with the numbers 0 through 9 – i.e. hard-code just the first value (0), then calculate the rest one at a time, storing them in the array as you go.
After you've stored them all, grab the ***seventh*** value (which would be 6) and store it in R2.

As always, step through your program and examine the values as they are stored in the array, and examine the final value stored in R2, to make sure your program works as expected.

3.2 What a character!

Exercise 02
You'll notice that Exercise 01 didn't ask you to output to console the array you built.
Why not?
Because as you know by now, numbers are not digits!

So now go and modify the output loop you just added, to make it output the *characters* corresponding to the *numbers* stored in your array, just as you learned to do in assignments 2 & 3.

3.3 Bits and pieces ...

Exercise 03
Let's try another modification of our well-used array program from Exercise 01:

This time, instead of calculating and storing the numbers from 0 to 9 in the array, calculate and store the first ten *powers of 2*, starting with 2^0 == 1
Finally, grab the seventh value (2^6) from the array, and store it in R2.

In order to do this, you will have to figure out how to calculate powers of 2.
Some hints:

- Mathematically speaking: How do I obtain 2^{n+1} if I have 2^n?
- What LC3 operation could I use to multiply a number by 2?

As always, step through your program and examine the values as they are stored in the array, and examine the final value stored in R2, to make sure your program works as expected.

I'm guessing you've already figured out that you can't simply output the values in the array to the console “as is”, so we have to manipulate them somehow to turn them into characters.

But this time all but the first four are *multi-digit numbers* when represented as decimal numbers, so our trick from the last exercise won't work – it can only convert the numbers from 0 to 9 into the single-digit ascii characters '0' through '9'.

This is going to take some more effort to solve, so we'll spread it out over a couple of labs & assignments.

Exercise 04

The first step will be to output the 1's and 0's of a stored value as *16 ascii characters*:

But wait! … You already did this in your last assignment!

I. Modify your code so that it prints out the “word” in register 2.

II. Now combine your code with ex3: drop it in after you load the 7th array value to R2.
At this point your program should print out 2^6 in binary format:

```
b0000 0000 0100 0000
```

So now we know how to print any number in binary format (put the value R2 and paste the binary print code); and we also know how to write a code that can build and traverse an array. It is time to combine our knowledge to print out all elements in the array!!

III. Modify your program so that it prints all the values in the array you created in ex3.
e.g.,

```
b0000 0000 0000 0001
b0000 0000 0000 0010
b0000 0000 0000 0100
         ….
         ….
b0000 0001 0000 0000
b0000 0010 0000 0000
```

Note: you do not need to add the spaces and the 'b', but if you have time you should try.

Keep your code clean and well organized, for example in pseudo-code it should look something like this:

```
prepare the binary array
for each element in array (hmm, this sounds familiar, see ex2)
    load the value to R2
    print the value in R2 (use 'the code')
end of program (HALT)
// program data
```

Exercise 05

Up until now, you have written all your code in a single block, just as you did in your first few weeks of C++
But in C++, you very soon learned how to package your code up as functions, and by now you couldn't even imagine writing a program of more than a dozen lines or so without using functions.

Well, Assembly Languages have their own version of functions, called subroutines (in fact functions and methods in HLLs are really just AL subroutines under the hood, as we'll learn in more detail in week 10).
We'll get to real subroutines next week, but for this last exercise, we'll make a “toy” one.

You have already learned about one control instruction - the conditional branch (BR)

There is another with a different memory addressing mode, the unconditional branch (JMP).
The syntax for JMP is

`JMP BaseReg`

where BaseReg is the “base register” (one of the 8 GPRs)
This instruction transfers control to the instruction located at address (BaseReg)

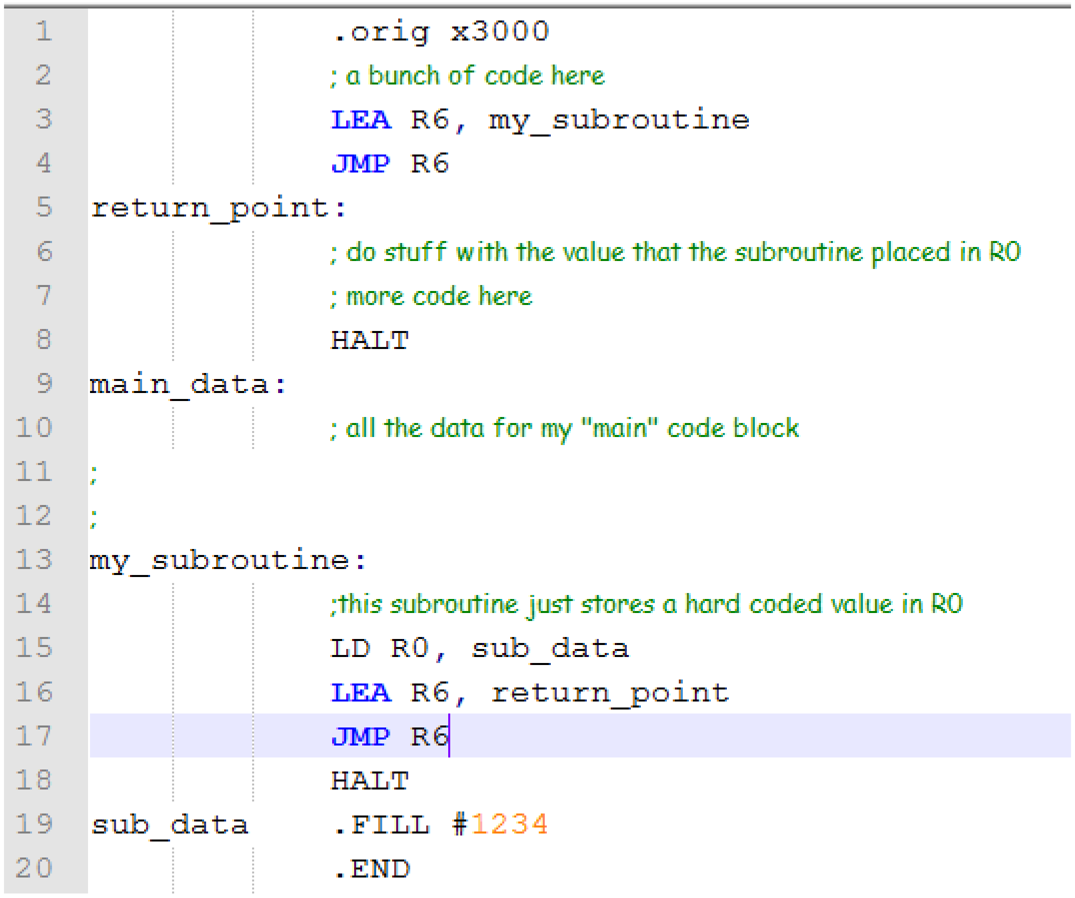
e.g. If R6 holds the value x3010, the instruction

`JMP R6`

will “jump” to the instruction at address x3010

Take a moment to follow the links to the instruction tutorials (and/or the descriptions in Appendix A of the text) to make sure you understand both BR and JMP.
For instance, how would you use the BR instruction to execute an unconditional jump?
How does this differ from JMP?

So you now know enough about JMP to use it to create a simple “code package”



```
                .orig x3000
                ; a bunch of code here
                LEA R6, my_subroutine
                JMP R6
return_point:
                ; do stuff with the value that the subroutine placed in R0
                ; more code here
                HALT
main_data:
                ; all the data for my "main" code block
;
;
my_subroutine:
                ;this subroutine just stores a hard coded value in R0
                LD R0, sub_data
                LEA R6, return_point
                JMP R6
                HALT
sub_data        .FILL #1234
                .END
```

(There is no real need for the HALT in line 18 - why not? I put it in mainly for the human reader).

Make sure you understand exactly what's going on here before you move on.

Now go back to your program of Exercise 04, and incorporate the bit-pattern outputter as a “code package” as shown above (of course, the bit-pattern outputter won't store anything anywhere - it is the equivalent of a C++ “void” function, aka procedure).
Where will you put the “calling code”?
Where will you put the “code package”?.

Make sure to include a newline output either in your loop, or in the “code package” so that each of the ten powers of two is output to the console on a line by itself:

```
b0000 0000 0000 0010
b0000 0000 0000 0100
        etc.
```

3.4 Man, I am hot! I rock at Assembly Language programming!!

sure, sure – but just make sure that you really have mastered the following skills and concepts:

- You should be able to do counter and sentinel controlled loops in your sleep
- Ditto for multi-way branches (the AL version of if statements)
- The difference between a *number* (an abstract concept) and its various *representations* – in different bases (2, 10, 16), and as a character (specifically, a numeric digit) with an ascii code
- Output values from 0 to 9 as their corresponding ascii characters
- Use the above techniques to inspect each digit of a 16-bit number and output it as the character '1' or '0'