SCC.111: GOODIES AND BADDIES - CONTINUED

You are expected to continue working on your project based on the **GameArena** library. Checkout last week's instructions.

[OPTIONAL] PRACTICE GIT AND SHARE YOUR PROJECT

We have created a public Gitlab repo (repository):

https://scc-source.lancs.ac.uk/scc.Y1/scc.111-workarea

Your goal is to create your first Gitlab account and copy this repo to your own profile. We will be giving more `git` examples in the first lecture this week.

Step 1. Create your Gitlab account

Create a Gitlab account and generate a new access token. The access token will be used instead of your password in the terminal. Alternatively, create your repo first and *create an access token just for that project.*

Step 2. Fork: Copy the repo to your own GitHub profile.

A *fork* is a copy of a repository. Forking a repository allows to freely experiment with changes without affecting the original project.

Navigate to this link and click **Fork Project** to fork the *GameArenaSubmissions* to your own repo.

Run the following command to store your credentials next time you use `git` .

```
git config --global credential.helper store
```

Clone the repo from your profile to the local disk:

```
git clone https://scc-source.lancs.ac.uk/yourusername/GameArenaSubmissions
```

Note: replace "*yourusername*" with the username you choose when creating your Gitlab account.

This will create a local folder in the current directory. The folder will have the same name as the repo, and will contain the **GameArenaSubmissions** files coming from the repo being cloned.

> [!IMPORTANT] Create a subfolder in the **scc.111-workarea** folder
> The subfolder should be named following this template: FIRSTNAME_LASTNAME

Now add your project files to the *scc.111-workarea/YOURNAME* folder. Then push the changes by running the following commands one-by-one:

```
git status
git add YOURNAME/*.java
git commit -m "project submission"
git push
```

NOTE: Replace *YOURNAME* with the folder name you created based on FIRSTNAME_LASTNAME template. Update the commit message to your liking.

Keep syncing your submission when you make significant updates to your project using the **same sequence**.

When asked to authenticate, use your github username as **username** and the token you generated in Step 1 as **password**.

Navigate to your repo in the browser to see the changes.

SCC.121: SORTING ALGORITHMS

SHELL SORT

Shell sort is a generalization of both bubble sort and insertion sort. In this sorting algorithm, you start by sorting pairs of elements that are far apart, then progressively sort closer and closer pairs of elements. Shell sort takes **two inputs**: an **input array**, to be sorted, and a **gap sequence array**, whose last element must be 1.

More concretely, you are given a sequence of gaps, say {5,3,1}. Then, you run a **gapped insertion sort** on the input array: essentially insertion sort, but only comparing elements that are exactly 5 indices apart. This will partially sort the input array. After which, you run another gapped insertion sort, now comparing elements exactly 3 indices apart. And finally, you run one last insertion sort, but now this is the traditional insertion sort.

- Code shell sort in Java.
- Why is shell sort always correct?
- Motivate why shell sort can have good time complexity, depending on the gaps sequence and the input array.

BINARY RADIX SORT

All sorting algorithms presented up to now (which includes the above shell sort) are **comparison-based sorting algorithms**. Other sorting algorithms, such as binary radix sort, do not use comparisons to solve the sorting problem.

Brief description of binary radix sort: We describe (most significant bit) binary radix sort on entries of length w (or also integers in $\{0, 1, \ldots, 2^w - 1\}$).
The algorithm takes the input array and partitions it into two sub-arrays, one in which the most significant bit of all entries is 0 and the other in which that most significant bit is 1. Then, each sub-array is sorted recursively and the sorted sub-arrays are concatenated to obtain the output array. For example, the radix sort algorithm (for $w = 6$) partitions [8,45,12,37,6,25] into [8,12,6,25] and [45,37] since $8 = 001000$, $45 = 101101$, $12 = 001100$, $37 = 100101$, $6 = 000110$ and $25 = 011001$.

You have to:

1. Implement a simple and not in-place recursive binary radix sort algorithm in Java.
2. Show that the worst-case time complexity of binary radix sort is $O(nw)$.
3. How does that worst-case time complexity compare with the $O(n \log n)$ worst-case time complexity of merge sort? You can take two use cases: (1) random permutations, and (2) large arrays of integers between 0 and 120 (age).

SCC.131: MULTIPLICATION GAME WITH ASSEMBLY

GOALS

Calling assembly-implemented functions in C programs can offer several benefits, particularly in terms of performance, low-level system access, and specific hardware control. Assembly language allows for highly optimized code that can be more efficient than code generated by C compilers. Modern processors include various specialized instructions (e.g., SIMD instructions, cryptography instructions) that may not be fully exploited by the C compiler or may require inline assembly to use effectively. The most important aspect for this module is that writing code in assembly helps you understand how higher-level constructs map to lower-level machine operations. This deep understanding can aid in better optimizing high-level code even when not directly writing in assembly.

This lab aims to get you familiar with coding ARM assembly functions and calling them in CoDAL programs. By the end of this lab, you should be able to:

- ☐ Implement an assembly function.
- ☐ Call Assembly from C.
- ☐ Implement basic algorithms in Assembly.
- ☐ Understand how the stack works.

The plan

For this exercise, you will implement a simple multiplication game with your micro:bit device. The main goal of this exercise is to implement a complex algorithm in ARM assembly. We will also practice how to call assembly functions from C++.

You will initially need to implement a multiplication algorithm in Assembly. For the first step, you will use the CPUlator platform to validate code correctness. After that, you will need to code in C++ a minimal micro:bit interface that will allow users to generate random numbers and then multiply them. The multiplication logic will be realized in assembly, using an algorithm called **Ethiopian Multiplication** which

uses shift operations and additions to multiply any two positive numbers.

TASK 1: ETHIOPIAN MULTIPLICATION IN ASSEMBLY

The Ethiopian multiplication algorithm is an ancient method of multiplying two numbers using a process that breaks down multiplication into a series of additions and bit shift operations. It's based on the principle that multiplying by 2 (doubling) and adding numbers can achieve the same result as traditional multiplication. This algorithm is particularly interesting for assembly because it leverages basic operations such as addition, subtraction, bit shifting, and conditional branching, which are well-suited to low-level hardware manipulation.

The multiplication algorithm involves the following steps:

1. **Write down the two numbers to be multiplied** at the top of two columns.
2. **Halve the number in the first column and double the number in the second column**, writing the results below the original numbers.
3. **Continue doubling and halving** each subsequent number, writing the results below the previous ones, until the number in the first column is reduced to 1.
4. **Strike out** every row where the number in the first column is even.
5. **Sum up** all the numbers that are left in the second column. This sum is the result of the multiplication.

For example, let's multiply the numbers 17 and 34:

A	B
17	34
8	~~68~~
4	~~136~~
2	~~272~~
1	544

$$17x34 = 34 + 544 = 578$$

You can also view a nice video from the BBC explaining the algorithm.

In order to help you better understand the algorithmic part of this exercise, we provide below a C implementation of the algorithm.

```c
#include <stdio.h>

// Function to perform Ethiopian Multiplication
int ethiopian_mult(int a, int b) {
    int result = 0; // Initialize result to 0

    // The while loop stops when a == 0.
    while (a >= 1) {
        if (a % 2 != 0) { // If 'a' is odd, add 'b' to the result
            result += b;
        }
        a = a >> 1; // Halve 'a'
        b = b << 1; // Double 'b'
    }

    return result; // Return the multiplication result
}
```

Your task at this stage is to implement the Ethiopian multiplication algorithm in ARM assembly. We strongly recommend you implement your code in the form of an ARM function that accepts a single input argument, which is the address of an integer array. You can hardcode an integer array for this step in the .data section of your program and use some hard-coded values. **Please revise the conventions for Assembly function implementation and make sure your function follows these conventions.** Below you can find a list of key Assembly Instructions and Concepts:

- `mov` : To load values into registers.
- `lsl` and `lsr` : Logical shift left and right instructions, useful for doubling and halving numbers.
- `add` : To sum up values.

- `cmp` and `beq` : Compare instructions and branch if equal, useful for checking if a number is even (test if the least significant bit is 0) and skipping the addition step.
- `b` : Unconditional branch instruction for loops.

A good template to start your code is the following:

```
@ source/eth_mult.s
    .syntax unified
    .data
    val: .word 14, 16

    .text
_start:
    @ TODO: Load the address of the array into register r0

    @ Call the function eth_mult
    bl eth_mult
    @ Make the function loop indefinitely
    b _start

    .global eth_mult
eth_mult:
    @ TODO: Implement your Ethiopian multiplication algorithm here
    bx lr
```

TASK 2: CODAL SETUP

The CoDAL build environment and the GCC compiler provide out-of-the-box support to build and link ARM assembly with C/C++ code. To start the integration job, you must start a new CoDAL project by re-downloading the git repo. Below you can find some command line commands to clone the CoDAL example repo to the folder `~/h-drive/microbit-asm/` (You can safely store the project in any location on your computer).

```
cd ~/h-drive/
git clone http://scc-source.lancs.ac.uk/scc.Y1/scc.131/microbit-v2-samples.git
↪  microbit-asm
cd microbit-asm/
```

Start in the background a build process by running the command `python3 build.py` on your command line terminal while you are in the micro:bit folder.

> **Note:** If you want to add support for CoDAL on your personal devices, you can find an install guide on the lab moodle page. The guide will help you set up the CoDAL build environment on your personal computer (details for Windows, OSX, and Linux). *We cannot guarantee to support personal computer setups, so please be aware that the lab computers are the only platforms that we will guarantee that things will work.*

Let's now create the template code for our project. We firstly need to modify the `source/main.cpp` file.

```
// source/main.cpp
#include "MicroBit.h"

MicroBit uBit; // The MicroBit object

// --- DECLARE GLOBAL VARIABLES, STRUCTURE(S) AND ADDITIONAL FUNCTIONS (if needed)
↪  HERE ---
int val[2] = {1};

// This links your Assembly function with the CoDAL runtime.
extern "C"
{
    void eth_mult();
}

// Event handler for buttons A and B pressed together
```

```
void onButtonAB(MicroBitEvent e)
{
    // DEVELOP CODE HERE
    eth_mult();
}

// Event handler for button A
void onButtonA(MicroBitEvent e)
{
    // DEVELOP CODE HERE
    eth_mult(int val[]);
}

// Event handler for button B
void onButtonB(MicroBitEvent e)
{
    // DEVELOP CODE HERE
}

int main()
{
    // Initialise the micro:bit
    uBit.init();
    // Initialize the random number generator
    uBit.seedRandom();

    // Ensure that different levels of brightness can be displayed
    uBit.display.setDisplayMode(DISPLAY_MODE_GREYSCALE);

    // Set up listeners for button A, B and the combination A and B.
```

```
    uBit.messageBus.listen(MICROBIT_ID_BUTTON_A, MICROBIT_BUTTON_EVT_CLICK,
↪   onButtonA);
    uBit.messageBus.listen(MICROBIT_ID_BUTTON_B, MICROBIT_BUTTON_EVT_CLICK,
↪   onButtonB);
    uBit.messageBus.listen(MICROBIT_ID_BUTTON_AB, MICROBIT_BUTTON_EVT_CLICK,
↪   onButtonAB);

    // You probably do not need to do anything after this point in the main method.

    // Enter the scheduler indefinitely and, if there are no other processes running,
    // let the processor go to a power-efficient sleep WITHOUT ceasing execution.
    release_fiber();
}
```

The code above provides the scaffold to manage three events: pressing button A (`onButtonA`), pressing button B (`onButtonB`), and pressing buttons A and B simultaneously (`onButtonAB`). Additionally, it registers your eth_mult function implemented in Assembly, and will execute it every time you press buttons A or A and B simultaneously. You should implement code to add functionality to the three event handlers.

You should also save the assembly code you implemented in step 1 in a new file called `source/eth_mult.s` . The build system will automatically detect and build the new file in the source folder and link the code with the C++ code.

TASK 3: INTERFACE

You should hopefully have now a working implementation of the Ethiopian Multiplication code in Assembly. In this step, you need to implement a basic interface with the micro:bit to generate random numbers and feed them to your Assembly function. Here is a breakdown of the interface that you need to implement:

- Press button A to generate a random number between 1 and 99 for the variable “a” and to show it on the LED display.
- Press button B to generate another random number between 1 and 99 for “b” and to show it on the LED display.

- Press buttons A and B simultaneously on the micro:bit to find out what the product is - that's what the answer would be if the numbers "a" and "b" were multiplied together. This part of the program should use an Ethiopian multiplication algorithm implemented in assembly.
- You can use this project in a competitive two-player game, where the two random numbers are read out, and each player must shout out the correct answer first to win a point.

You already have a set of 3 event handlers to manage different button press events and a global array variable `val` which can hold up to two elements and which you should use to store your random numbers. Your micro:bit implementation should generate a random number when buttons A or B are pressed, store the result in the first or second array element in variable `val` respectively, and print the number values using the LED screen.

Random number generation in CoDAL is a **bit** different from what you are used to in C. The runtime provides the following function:

```
int microbit_random(int max);
```

The function will return a random integer number between the values [0, max]. You can use this function to generate a and b values, but you probably want to skip 0 values, as the result will be zero.

For your number displaying functionality, you can use the LED screen and print values using the method:

```
uBit.display.print(int a);
```

TASK 4: CALLING ASSEMBLY FROM C++

In this final step, we will apply your knowledge of memory instructions and functions in assembly, in order to implement the functionality that will pass the variable val into your assembly Ethiopian multiplication function and return the multiplication result to your C++ code. It is worth focusing a bit on the C++ code below:

```
// This links your Assembly function with the CoDAL runtime.
extern "C"
{
```

```
    void eth_mult();
}
```

This line informs the compiler that a C function is available in another file that implements a function called eth_mult. The function accepts a single array pointer variable as input and will be returning an int variable. We also included the following line in our .S file:

```
.global eth_mult
```

This line tells the assembler that the eth_mult function is a global symbol and can be accessed from other files. The assembler will create a record of this function in the `.o` file and the linker will be able to link the function with your C++ code.

Another thing to look for in your implementation is how the `val` variable is passed to your Assembly function. The parameter will effectively be a pointer to the address of the first element of the array. The pointer will point to an address in the data section. You can use that value as the second operand in the memory instructions **str** and **ldr**, to store and load data from main memory. The C++ assembly code generated by the compiler will be responsible for loading the address of the array into register `r0` before calling the function. If you implemented your code to read the value from an array in Stage 2, then you shouldn't need to change anything in your Assembly code. In your C++ code, you should be able to call the function by using the following line:

```
eth_mult(val);
```

At this stage, it is good to revise the material about functions in Lecture 6 of the assembly part of the course and figure out how parameters are passed into a variable and how results can be returned from an assembly function.

HACKER EDITION

SCC.121: MORE ON SORTING ALGORITHMS

1. Code **heapsort** in java. You can find the description of this sorting algorithm on wikipedia.

2. The following problem is for the more mathematically inclined, and **you won't be required to know any of this for the SCC 121 quiz or exam**. Prove that the average-case time complexity of quicksort is $O(n \log n)$. You can assume that the array contains no duplicates and that entries of the input array are integers in $\{1, \ldots, n\}$.

Hints for (2): - To do so, you can first prove that the average-case complexity of quicksort satifies the recurrence relation: $T(n) = c \cdot n + (1/n) \cdot \sum_{i=0}^{n-1}(T(i) + T(n-i-1))$, for some constant $c > 0$. (For that, note that the last entry of a random permutation (which is the input) is a uniformly random integer in $\{1, \ldots, n\}$.) - To get the average-case time complexity from the recurrence relation, you may need the following: $\sum_{i=1}^{n-1} 1/i = \int_1^n \lfloor 1/x \rfloor dx < \int_1^n (1/x) dx = \ln n$.

SCC.131: MULTIPLICATION GAME WITH ASSEMBLY

Task 5: Negative number support

The algorithm you have implemented for your Ethiopian multiplication can perform operations with positive numbers. Can you extend your code to support negative numbers?

> **Hint**: A good way to approach this task is to convert negative numbers to positive using the two's complement conversion algorithm and perform the multiplication. Don't forget though to convert your result to a negative value, if exactly one of the input numbers was negative.

Task 6: Inline Assembly

Your project implements your Assembly as a function, which is invoked by the C++ runtime. In lecture 9 we discussed the mechanism of inline assembly. Can you convert your function invocation into handwritten inline Assembly?