Problem 2: GodBolt *[24 points]*

Scribe: **[Scribe's name here]**

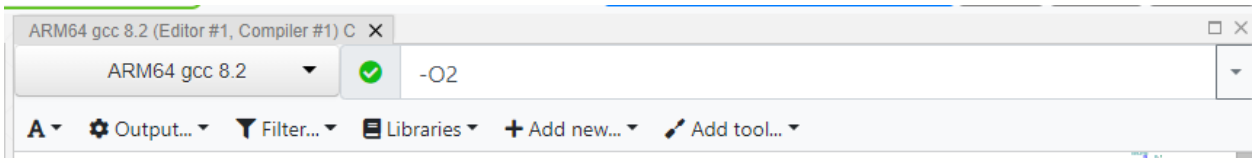
Go to godbolt.org. This is a website that will compile source code from several different programming languages into one of many target assembly languages, and color code the correspondence between the two programs.

In the left hand window, select "C" is the source language and paste in the following code:

```c
#include <stdlib.h>
void scale_array(unsigned long long *arr, unsigned long size) {
    int i=0;
    while(i<size) {
        unsigned long long temp = arr[i];
        temp *= 100;
        arr[i] = temp;
        i++;
    }
}
```

Your window should look like this (with perhaps rows highlighted as different colors):

In the right hand window,select "carm64g820" or “ARM64 gcc 8.2" for the architecture and "-O2" for the compiler options:



After a moment, the right window should be populated with ARM assembly code. You can see which C statements each instruction corresponds to by matching the colors between the two windows.

Because this compiler is using the full Arm assembly language (not just the LEG subset described in lecture), there are a few instruction variants you may be unfamiliar with (note that we do **NOT** expect you to be familiar with these instructions outside the scope of this problem). The first is:

```
add x1, x2, x3, lsl 4
```

This instruction combines an `add` instruction and an `lsl` instruction, which could be translated to:

```
x1 = x2 + (x3 << 4)
```

The second new instruction looks like:

```
str x0, [x1], 4
```

This instruction simply combines a store instruction with an `add` instruction. It is equivalent to:

```
sturw x0, [x1,#0]
add x1, x1, #4
```

1. In the space, below paste the compiled assembly and explain how each instruction is implementing the specified C code. In particular, describe how the assembly multiplies an element by 100 without the use of an explicit multiply instruction. *[12 points]*

```
        cbz     x1, .L1             // branch to end if size is zero
        add     x1, x0, x1, lsl 3   // x1 points to end of array (&arr[size])
.L3:                                // start of loop
        ldr     x3, [x0]            // x3 = arr[i]
        add     x2, x3, x3, lsl 1   // x2 = arr[i]*3
        add     x2, x3, x2, lsl 3   // x2 = x2*8+arr[i] (i.e. arr[i]*25)
        lsl     x2, x2, 2           // x2 = x2*4 (i.e. arr[i]*100)
        str     x2, [x0], 8         // arr[i] = arr[i]*100; i++
        cmp     x0, x1              // if at end of array...
        bne     .L3                 // ...don't execute loop again
.L1:
        ret                         // return
```

Change the source code above so that rather than multiplying each element of the array by 100, it instead multiplies each element by 127.

2. Briefly (30 words or fewer) describe the change in the resulting assembly and how it accomplishes the multiplication. [*12 points*]

   Rather than doing several shifts and adds, multiplying by 127 can be done by shifting to the left 7 (i.e. multiplying by 2^7=128) and then subtracting the original value once (i.e. x*127 == x*128 -x).

Problem 3: Putting it All Together *[24 points]*
Scribe: **[Scribe's name here]**

Convert the following C code to ARM. Assume the system requires memory alignment, and that the assembler will not fix opcodes. Use labels when possible. The return value should be in register *X2*. Assume:

- Starting address of the **data** array is stored in register ***X0***, and consists of **512 elements.**
- Assume all operations will be within the **32 bit unsigned integer limits.**

Code explanation (you don't need to read this to answer the question):
The ABC370 factory is responsible for producing packs of 20 pencils. However, this factory is not always perfect, which means some packs will contain less than the desired amount of 20 pencils. In the case where there are less than 20 pencils per pack, we simply multiply by 4. Return the number of pencil packs not changed in register X2.

These questions will help you fill out the below *[10 points]:*

1. How much padding (in bytes) is needed between the `pack_id` array and `num_penc`?
   4 bytes

2. How much padding is needed at the end of the `pencil_pack` struct?
   0

3. How large is each `pencil_pack` struct instance?
   32 bytes

4. Where can the address of "data" be found at the start of the `check_pencil_arr`, assuming we're following the ARM ABI? X0 (from question description)

5. Given the skeleton code below, determine what register is holding:
   a. `i` X3
   b. `good_count` X2 (from question description)
   c. `data[i].num_penc` X5

C	ARM

```
struct pencil_pack{
    char pack_id[20]; //20B
    int64_t num_penc; //8B

};

int32_t
check_pencil_arr(
           struct pencil_pack data[]) {
  int32_t good_count = 0;

  for(int32_t i = 0; i < 512; ++i){
     if(data[i].num_penc < 20){

         data[i].num_penc *= 4;

     } else {
        ++good_count;
     }
  }

  return good_count;
}
```

```
func:
     MOVI    X2,   #0x00
     MOVI    X3,   #0x00
loop:
     CMPI    X3,   #512
     B.EQ    end
     LSL     X4,    X3,   #5
     ADD     X4,    X4,    X0
     LDUR    X5,   [X4,   #24]
     CMPI    X5    #20

     B.LT    smaller
     ADDI    X2,    X2,   #1
     B       forend
smaller:
     LSL    X5,    X5,   #2
     STUR   X5    [X4,   #24]
forend:
     ADDI    X3    X3   #1
     B   loop
end:
```

Note:

- Indexing into an array can be thought of as pointer arithmetic – **arr[i] = *(arr + i)**
- Since each struct takes 32 bytes, this means the **ith** element of pencil_pack is at a memory address of **pencil_pack (base address) + 32 (struct size in Bytes) * i (iter)**
- From lab 2, we know that a << b = a * 2^b (i.e. **a * 32 = a * 2^5 = a << 5**)