name:	Vince Rothenberg
course:	csci 10
assignment:	homework 13
prepared:	Tue, May 5, 2020 // 9:25 am

1. What is the purpose of the .macro and .endm GNU AS directives?

The commands .macro and .endm allow you to define macros that generate assembly output.

2. Write the GNU AS code to define a macro named addOne that increments the register r0 by 1.

```
.macro addOne

    add r0, 1

.endm
```

3. Write the GNU AS code to define a macro named resetCounter that test the value in r1, and sets the value in r1 to 1 if the current value of r1 is 100.

```
.macro resetCounter

    teq        r1, 100
    it           eq
    moveq r1, 1

.endm
```

4. Can you pass arguments to GNU AS macros? Can these arguments have default values?

Yes, you can pass arguments to GNU AS macros.

Yes, you can supply a default value for any macro argument by following the name with â€˜ = deflt â€™.

5. What is the purpose of the .include GNU AS directive?

This directive provides a way to include supporting files at specified points in your source program. The code from file is assembled as if it followed the point of the .include; when the end of the included file is reached, assembly of the original file continues.

6. On ARM processors, how many bytes are needed for an array of 25 integers? Write the ARM assembly code to declare an array named numbers of 25 integers.

An integer is 32 bit wide, which is 4 bytes. An array of 25 integers would require 4*25 = 100 bytes.

```
.data  // Arrays declared in data section of code
.balign 4
numbers: .skip 100
```

7. What is the purpose of the GNU AS directive .skip? Describe .skip in detail.

```
.skip     size ,     fill
```

This directive emits size bytes, each of value fill. Both size and fill are absolute expressions. If the comma and fill are omitted, fill is assumed to be zero. This is the same as `.space'.

8. Write the ARM assembly instructions to (1) load the address of numbers (using =; assume the array declared above) into r0, (2) load the immediate value 1 into r1, and (3) store the value 1 as the first element of the array.

```
ldr      r0, =numbers
mov   r1, 1
str      r1, [r0]  // numbers[0] = 1
```

9. Briefly describe the non-updating indexing modes in ARM.

Non-updating indexing mode adds (or subtracts) the immediate value to form the address.

```
// Examples of non-updating indexing mode
mov r2, #3          /* r2 â†• 3 */
str r2, [r1, #+12]  /* *(r1 + 12) â†• r2 */

// We want to populate array with 10, 20, 30, 40, 50, ...
ldr        r0, =numbers
mov     r1, 10
str        r1, [r0, #0]   // numbers[0] = 10

mov     r1, 20
str        r1, [r0, #4]   // numbers[1] = 20

mov     r1, 30
str        r1, [r0, #8]   // numbers[2] = 30

mov     r1, 40
str        r1, [r0, #12] // numbers[3] = 40

mov     r1, 50
str        r1, [r0, #16] // numbers[4] = 50

// NOT WORKING EXAMPLE
str        r5, [r3, #-8] // Must be deeper into the array to move backward.
```

10. Briefly describe the updating indexing modes in ARM.

In updating indexing modes a register is updated with the address synthesized by the load or store instruction.

```
// Examples

ldr       r0, =numbers

// We want to populate our array with 3, 6, 9, 12, 15, ...

mov        r1, 3
str           r1, [r0]  // numbers[0] = 3

add         r0, 4 // updated the base address so it 'appears' the array starts 4 bytes over,
mov        r1, 6
str           r1, [r0]  // numbers[1] = 6

add         r0, 4
mov        r1, 9
str           r1, [r0] // numbers[2] = 9

// Mistake
add        r0, 8  // moved marker by 8 bytes, skipped numbers[3] slot
mov       r1, 12
str          r1, [r0] // numbers[4] = 12
```