Homework Assignment 4

Due: Tue, 29 Sep 2020 23:59:59 (approximately 72 days ago)
[Score: 10 / 10 points possible]
Weight = 1.0

The goal of this homework is translate more complicated C functions and subroutines that contain stack-allocated data into ARM Cortex-M0 assembly language. You will also experiment with functions that have more than four arguments. This requires some additional understanding of how the stack works as well as how to reserve and free space on the stack. One of the easiest ways to interact with a microprocessor is using a *serial port*. This time, we give you some initialization code (stdio.s) that you can use to configure a serial port of the STM32. You can connect the serial port to your computer with a USB-to-serial adapter in your lab kit. Detailed instructions on how to use serial communication can be found on the serial port setup page.

The serial port page refers, generically, to pins on the STM32 development board as TX and RX. Many of the pins on the STM32 can take on these roles, depending on the configuration. For this assignment the stdio.s file configures the serial port to use PB10 for transmit (TX), and PB11 for receive (RX). You should cross-connect those pins to the conjugate pins on the USB-to-serial adapter. (I.e., connect the STM32's TX to the adapter's RX and the STM32's RX to the adapter's TX).

Before you plug in the USB-to-serial converter:

The single most important thing you should do is change the USB-to-serial converter to 3.3V operation. **It is set to 5V by default.** Move the shorting jumper on the USB-to-serial converter to the 3.3V setting. If you connect 5V to the wrong place on your microcontroller, you will irreversibly damage it. Read that section of the serial port page and check the picture.

More than four arguments

You've had much experience using subroutines that take at most four arguments that are passed in R0 - R3. Subroutines with more than four arguments are still called with the first four arguments in R0 - R3, and the excess arguments are passed on the stack. For example, consider the following function that accepts five arguments, and a main function that calls it:

Example

```
int addfive(int a, int b, int c, int d, int e)
{
        return a + b + c + d + e;
}

int main() {
        addfive(2,4,6,8,10);
}
```

Assembly Translation

```
.global addfive
addfive:
        adds r0, r1       // a += b
        adds r0, r2       // a += c
        adds r0, r3       // a += d
        ldr  r1, [sp,#0]  // Load e from the stack.
        adds r0, r1       // a += e
        bx lr             // Return
```

Call it like this:

```
.global main
main:
        movs r0,#10    // Argument 5 is 1
        sub  sp,#4     // Allocate 4 byte
        str r0,[sp,#0] // Store 5 on the
        movs r0,#2     // Argument 1 is 2
        movs r1,#4     // Argument 2 is 4
        movs r2,#6     // Argument 3 is 6
        movs r3,#8     // Argument 4 is 8
        bl addfive     // Call addfive()
        // IMPORTANT: Deallocate 4 bytes
        add  sp,#4
        wfi
```

You know that PUSH and POP make the stack grow downward and shrink upward, respectively. The amount of the change is proportional to the number of registers saved and restored---four bytes per register. We can do the same with subroutines, but must do so with one rule: Any called subroutine must restore the stack to the same it was when it was called. Similarly, the caller may allocate space on the stack before calling a function as it did in the example above.

This manner of passing more than four arguments is completely compatible with saving and restoring registers as well. We only need to keep track of how many additional space is allocated on the stack with the saved registers. For example, we could rewrite the example, above, with PUSH and POP, and we can (uselessly) save R4 -- R7:

Example

```
int addfive(int a, int b, int c, int d, int e)
{
        return a + b + c + d + e;
}

int main() {
        addfive(2,4,6,8,10);
}
```

Assembly Translation

```
.global addfive
addfive:
        push {r4-r7,lr}
        adds r0, r1       // a += b
        adds r0, r2       // a += c
        adds r0, r3       // a += d
        ldr  r1, [sp,#20] // Load e from the stack.
        adds r0, r1       // a += e
        pop  {r4-r7,pc}   // Return
```

Call it like this:

```
.global main
main:
        movs r0,#10    // Argument 5 is 1
        sub  sp,#4     // Allocate 4 byte
        str r0,[sp,#0] // Store 5 on the
        movs r0,#2     // Argument 1 is 2
        movs r1,#4     // Argument 2 is 4
        movs r2,#6     // Argument 3 is 6
        movs r3,#8     // Argument 4 is 8
        bl addfive     // Call addfive()
        // IMPORTANT: Deallocate 4 bytes
        add  sp,#4
        wfi
```

Allocating space on the stack for local variables

A caller allocates space on the stack for extra arguments. A subroutine can allocate space on the stack for a *local variable* such as an array. To do so, we need only keep track of how much space was allocated and deallocate it before returning. For example, consider the following example:

Example

```
int addfive(int a, int b, int c, int d, int e)
{
        int x;
        int sum = 0;
        int array[100];
        for(x=0; x < 100; x++)
                array[x] = x;
        for(x=a; x <= b; x++)
                sum += array[x];
        sum += a + b + c + d + e;
        return sum;
}
```

Assembly Translation

```
.global addfive
addfive:
        push {r4-r7,lr}
        sub  sp,#400      // Allocate 100 integers
        mov  r7,sp        // R7 is the beginning of "array"
        movs r5,#0        // Let R5 be "sum"
        movs r4,#0        // Let R4 be "x"
for1:
        cmp  r4,#100
        beq done1
        lsls r6,r4,#2     // R6 is 4*x
        str  r4,[r7,r6]   // array[x] = x
        adds r4,#1        // x++
```

Call it like this:

```
.global main
main:
        movs r0,#10    // Argumen
        sub  sp,#4     // Allocat
        str r0,[sp,#0] // Store 5
        movs r0,#2     // Argumen
        movs r1,#4     // Argumen
        movs r2,#6     // Argumen
        movs r3,#8     // Argumen
        bl addfive     // Call ad
        // IMPORTANT: Deallocate
        add  sp,#4
        wfi
```

```
int main() {
        addfive(2,4,6,8,10);
}
```

```
        b for1
done1:
        movs r4, r0       // x = a
for2:
        cmp  r4, r1       // x <= b?
        bgt done2
        lsls r6,r4,#2     // r6 = r4 * 4
        ldr  r6,[r7,r6]   // r6 = array[x]
        adds r5,r6        // sum += array[x]
        adds r4,#1        // x++
        b for2
done2:
        adds r5, r0       // sum += a
        adds r5, r1       // sum += b
        adds r5, r2       // sum += c
        adds r5, r3       // sum += d
        ldr  r1, [sp,#420] // Load e from the stack.
        adds r5, r1       // a += e
        movs r0, r5       // Want to return value in r5
        add  sp,#400      // Deallocate 100 integers
        pop  {r4-r7,pc}   // Return
```

One important thing to keep in mind with stack allocation is that the SP register can only be incremented or decremented by 4. If you wanted to make space for an array of characters such as **`char buffer[13];`** then you would need to round **up** the size of the allocation to **16**. The fundamental reason for this is that the lower two bits of the SP register are always zero. The special SUB and ADD instructions that work with SP also only work with values divisible by four.

Standard I/O on the STM32

One of the best reasons to use the serial port is that the STM32 Standard Peripheral firmware allows you to use it with the C Standard I/O library calls. The stdio.s functions set these up to work with the serial port. As you know, one of the most common C functions to call is **`printf()`**. Now, we can call it in assembly language. For instance:

```
.global main
main:
        bl serial_init
        ldr r0,=greeting
        bl printf
        wfi
greeting:
        .string "Hello, World.\n"
        .align 2 // Align anything after this.
```

Of course, the primary reason we use **`printf()`** is so that we can print the values of variables in a formatted manner. The firmware for the STM32 is no less capable:

```
.global main
main:
        bl serial_init
        ldr r0,=format
        movs r1,#34
        ldr r2,=0xdecafbad
        bl printf
        wfi
format:
        .string "The value of x=%03d.  Beverage preference is 0x%08x!\n"
        .align 2
```

Each of the following exercises are independent of the others, but you should write and test all of your functions in a single .s file in System Workbench. A code skeleton has been provided for you in the **`hw4.s`** file. The simulator won't do you a bit of good for this assignment since we don't have a serial port or terminal emulator for it (yet). These subroutines will be called by a C program, so you should **definitely** be sure that you don't overwrite R4 - R11 without saving them first. Each subroutine should be prefaced with a *.global subroutine_name* to make it visible to other modules. An example C that calls each function is provided for you in the **`main.c`** file. This time, when you are done, you can submit the entire **`hw4.s`** file that contains all of your subroutines.

You should assume, for each subroutine, that **`serial_init`** has already been called for you by **`main()`**. You don't have to do that in your code.

Q1: hello [1 point]

Translate ("hand compile") the following C function into assembly language. First, let's just make sure that Standard I/O is working and you have your serial port wired correctly. What we expect to see here is an assembly language subroutine that sets up the first and second arguments as strings and does a **`bl printf`**.

```
const char login[] = "xyz";
void hello(void) {
    printf("Hello, %s!\n", login);  // Here, printf is given two arguments
}
```

Q2: showmult2 [1 point]

Translate ("hand compile") the following C function into assembly language. More difficult this time. The first argument given to printf() is the string with the format statements. The second argument will be the value of the 'a' variable (which was passed in as argument 1). The fourth argument will be the product of the 'a' and 'b' variables. Make sure that the spacing in the format string exactly the way it is presented below.

```
void showmult2(int a, int b)
{
```

```
    printf("%d * %d = %d\n", a, b, a*b); // Here, printf is given four args
}
```

Q3: showmult3 [1 point]

Translate ("hand compile") the following C function into assembly language. Since there are five arguments, you'll need to put one of them in stack memory.

```
void showmult3(int a, int b, int c)
{
    printf("%d * %d * %d = %d\n", a, b, c, a*b*c); // five args
}
```

Q4: listing [1 point]

Translate ("hand compile") the following C function into assembly language.

```
void listing(const char *school, int course, const char *verb,
             int enrollment, const char *season, int year)
{
    // You need to allocate space on the stack to hold some of the
    // seven parameters to printf().  Then move the arguments around
    // to call printf() with the proper arguments.
    printf("%s %05d %s %d students in %s, %d\n",
           school, course, verb, enrollment, season, year);
}
```

Q5: trivial [1 point]

Translate ("hand compile") the following C function into assembly language. This function requires you to allocate a 100-word on the stack.

```
int trivial(unsigned int n)
{
    int tmp[100]; // a stack-allocated array of 100 words
    for(int x=0; x < sizeof tmp / sizeof tmp[0]; x += 1)
        tmp[x] = x+1;
    if (n >= sizeof tmp / sizeof tmp[0])
        n = sizeof tmp / sizeof tmp[0] - 1;
    return tmp[n];
}
```

Q6: reverse_puts [2 points]

Translate ("hand compile") the following C function into assembly language.

```
void reverse_puts(const char *s)
{
    unsigned int len = strlen(s);
    // Round up size to the next higher multiple of 4.
    int newlen = (len + 4) & -3;
    // this is a dynamically-sized stack-allocated array
    char buffer[newlen];
    buffer[len] = 0;
    for(int x=0; x < len; x += 1)
        buffer[len-1-x] = s[x];
    puts(buffer);
}
```

Q7: sumsq [3 points]

Translate ("hand compile") the following C function into assembly language.

```
int sumsq(unsigned int a, unsigned int b)
{
    int tmp[100];
    if (a >= 100)
        a = 99;
    if (b >= 100)
        b = 99;
    int step = 1;
    if (a == b)
        step = 0;
```

```
    else if (a > b)
        step = -1;
    for(int x=0; x < sizeof tmp / sizeof tmp[0]; x += 1)
        tmp[x] = x * x;
    int sum = 0;
    for(int x=a; ; x += step) {
        sum += tmp[x];
        if (x == b)
            return sum;
  }
}
```

Academic Honesty Statement [0 ... -10 points]

By typing my name, below, I hereby certify that the work on this homework is my own and that I have not copied the work of any other student (past or present) while completing it. I understand that if I fail to honor this agreement, I will receive a score of zero for the assignment, a one letter drop in my final course grade, and be subject to possible disciplinary action.

Raghuram Selvaraj 

Your hw4.s file

Turn in your hw4.s file here.

```
.cpu cortex-m0
.thumb
.syntax unified
.fpu softvfp

.global login
login: .string "selvara2"
hello_str: .string "Hello, %s!\n"
.align 2
.global hello
hello:
        push {lr}
        ldr r0, =hello_str
        ldr r1, =login
        bl printf
        pop  {pc}

showmult2_str: .string "%d * %d = %d\n"
.align 2
.global showmult2
showmult2:
        push {r4-r7, lr}
        movs r4, r0
        movs r5, r1
        movs r6, r4
        muls r6, r5

        ldr r0, =showmult2_str
        movs r1, r4
        movs r2, r5
        movs r3, r6

        bl printf
        pop  {r4-r7, pc}

// Add the rest of the subroutines here
showmult3_str: .string "%d * %d * %d = %d\n"
.align 2
.global showmult3
showmult3:
        push {r4-r7, lr}
        movs r4, r0
        movs r5, r1
        movs r6, r2
        movs r7, r4
        muls r7, r5
        muls r7, r6

        ldr r0, =showmult3_str
        movs r1, r4
        movs r2, r5
        movs r3, r6

        sub sp, #4
        str r7, [sp, #0]
        bl printf
        add sp, #4
        pop  {r4-r7, pc}

listing_str: .string "%s %05d %s %d students in %s, %d\n"
.align 2
.global listing
listing:
```

```
listing:

        push {r4-r7, lr}

        movs r4, r0
        movs r5, r1
        movs r6, r2
        movs r7, r3

        ldr r0, =listing_str
        movs r1, r4
        movs r2, r5
        movs r3, r6

        ldr r4, [sp, #20]
        ldr r5, [sp, #24]

        sub sp, #4
        str r5, [sp, #0]

        sub  sp, #4
        str r4, [sp, #0]

        sub sp, #4
        str r7, [sp, #0]

        bl printf
        add sp, #12
        pop  {r4-r7, pc}

.global trivial
trivial:
        push {lr}
        sub sp, #400
        mov r3, sp
        for1:
                movs r1, #0
        check1:
```