ENSC 254 – HW4 – bigAdd Assignment

June 2021 – Due July 1 – max 1 partner

Introduction

In the previous three labs, we have gotten familiar with how instructions are structured at a low level on an ARM7 processor, how registers interact with memory, and the use of the status register to provide conditional branching. We have also touched on how subroutines fit into the picture.

In this assignment, you will be creating and testing a subroutine that can perform arbitrary-sized unsigned addition. This subroutine will take arguments using the convention as appears in the ARM *Application Procedure Call Standard* (APCS) and, for other purposes, probably make use of something called a stack.

The Stack and the ARM APCS

Since our coding is getting a little more complicated, we are going to start using a program stack to save some register values at the beginning of a subroutine and restore those register values just before the subroutine returns. We are also going to pay closer attention to the ARM *Application Procedure Call Standard* (APCS). The APCS defines the standardized use of registers, the proper conventions for calling and returning from subroutines, and standardization of which register values need to be preserved by subroutines.

The stack is a structure in memory that can be used for purposes including preserving register values – allowing for a subroutine to save register values, overwrite the registers for its own calculations, and restore the registers later. From the point of view of a program running on an ARM processor, a stack is simply an array of values in memory with a single pointer, which moves as data is put onto and removed off of the stack. While any register can be used as a pointer to a stack, r13 is typically reserved for this purpose and also, in fact, r13 is mandated by the APCS.

The Store Multiple (`STM`) and Load Multiple (`LDM`) instructions are used to store multiple register values to and load register values from memory, including to/from the memory for the stack. There are multiple ways from which one can choose to operate a stack: have the stack grow to larger (Ascending) or smaller (Descending) memory addresses, and have the stack pointer normally point at the last item on the stack (Full), or the next empty space (Empty). For STM and LDM instructions, furthermore, the register pointing to memory can be updated or not – for purposes of a stack typically

the register is updated. This gives flexibility, but leads to complicated instruction formats.

For example,

- to store R4 through R6 and the LR to the stack and then read them back,
- using R13 (SP) as the stack pointer as is conventional for the stack,
- using a Full-Descending style for the stack, and
- updating (writing back to) the stack pointer as is needed for stack operations,

the following instructions are used:

```
stmdb r13!, {r4-r6, lr}; @ Store Multiple, Decrement SP before, update
;@ …
ldmia r13!, {r4-r6, lr}; @ Load Multiple, Increment SP after, update
```

To simplify the symbolic program and use more common names for interfacing with the stack, the pseudo-instructions `PUSH` and `POP` can be used instead of the `STMDB` and `LDMIA` instructions. These replace the full-descending style instructions using r13 (SP), described above, simplifying the lines of code to:

```
push  {r4-r6, lr}
;@ …
pop   {r4-r6, lr}
```

Note that it is common to push the link register to the stack but then pop the value back to the program counter. What effect would this have?

```
push  {r4-r6, lr}
;@ …
pop   {r4-r6, pc}
```

According to the APCS, registers r0-r3 are used for arguments passed to subroutines, while r0, and r1 if needed, is/are used for a returned value. The values in registers r0-r3 and r12 need not be preserved by a subroutine. We will generally stick to this standard when passing arguments to – and data back from – our subroutines.

The APCS also states that r4-r11 can be used for automatic variables (though r9 is apparently special in some less-used versions of the APCS). If a subroutine will modify any of these registers, the original values for applicable registers must be preserved by the subroutine, typically by pushing them onto the stack, so that the registers can be later restored before returning from the subroutine. To overcome the issue with nested subroutine calls, the Link Register should also be stored on the stack if a nested subroutine call will be made.

Section 5.4 of *Modern Assembly Language Programing with the ARM Processor* and Chapter 13 of *ARM Assembly Language: Fundamentals and Techniques (2nd Edition)*, as well as the *Procedure Call*

Standard for the ARM Architecture document (see Canvas) can be referenced for more details.

Exercise – bigAdd Subroutine

In the "studentWork" project for Keil uVision, fill in the bigAdd subroutine in the file bigAdd.S. The purpose of bigAdd is to do arbitrary-sized unsigned additions. One use of bigAdd can be to create a Fibonacci routine that can accurately calculate the N^{th} Fibonacci number for large values of N. So N=1000 should be no problem for example. When being called, bigAdd takes three arguments in r0, r1, and r2. r2 contains an argument called `maxN0Size` as described below. r0 and r1 point to arrays of 32-bit unsigned integers. Element 0 of each array stores the number of further words that contain valid data in the array. For example, if element 0 contains the value 23, then the following 23 words contain valid data. The least-significant word comes first, so this is a little-endian style. Note that, now, the template project uses a little-endian version of the ARM7 processor (see https://en.wikipedia.org/wiki/Endianness). The arbitrary-sized unsigned data pointed to by r1 is added to the arbitrary-sized unsigned data pointed to by r0 and, in the absence of an error, the result/sum ends up in the arbitrary-sized unsigned data initially pointed to by r0. If element 0 of an array contains 0, then the value of the arbitrary-sized number as a whole is, by our definition, 0.

Sometimes a carry out in the final "partial" addition will cause a developing sum using, up to this point, N valid words to increase in size such that it requires (N+1) words to be fully stored.
For example (written for convenience in "big-endian" style – the words are actually reversed in memory in the little-endian style):

0x80000000,00000001,00000002+

0x80000003,00000004,00000007

0x00000001,00000003,00000005,00000009

This is all good and well, but what if there is no available space in memory for the extra word in the expanded result integer where that result integer is supposed to go?

I am proposing a subroutine that conforms to the following c-language function signature (using the provided typedef):

```
typedef unsigned int bigNumN[];
// returns -1 for error in inputs, 1 if overflow/carry-out and 0 if no overflow/carry-out
int bigAdd(bigNumN bigN0PC, const bigNumN bigN1PC, unsigned int maxN0Size);
```

where `maxN0Size` specifies the maximum number of valid words that can fit in the `bigN0PC` array (that is, the array size less one because word 0 in the array stores the number of valid words). The subroutine returns 1/0/-1 in r0 as indicated in a comment above the function signature. `maxN0Size` should always be greater than or equal to the maximum of the valid words in `bigN0PC` or `bigN1PC`. If it is not, you should return -1 indicating an error condition.

Here are some examples:

Consider `bigN0PC` has 1 valid word and `bigN1PC` has 2 valid words. If `maxN0Size` is 1 then that is an

error so in register r0 return -1 from the subroutine. If `maxN0Size` is 2 then there is no error, and the return value may be 0 or 1 depending. If `maxN0Size` is 3 in this case then the return value of the subroutine (in register r0) will always be 0 because 3 words is always large enough to hold whatever sum is possible from an addition with operands of less than 3 words (like the 1- and 2-word operands specified in these examples).

The template project for this assignment contains testing code to test your bigAdd subroutine. It calls a library function called *memcmp* to compare values in memory. The linker links *memcmp* to our code. This is the same *memcmp* that can be used from C-language. Parameters to *memcmp* are passed in order in r0 through r2, and the return value is returned in r0.

Please add many more rows of data to the testing table so that you are doing thorough testing of your subroutine. You may also modify things to add more information for each testcase. Note, element 0 of each input array for a given test needn't contain the same value.

There might be a prize for the pair of students who come up with the fastest implementation for the bigAdd subroutine. Details to be provided if we go ahead with this. If there is a tie, the subroutine taking less space will break the tie. As mentioned in class, we expect that every student will work with a partner on this assignment.

To demonstrate that the testing code works, I plan to also provide a project with a C++ implementation of **`bigAdd`**. In your implementation, make use of the carry flag in the status register when doing additions. In C and C++ as defined by the language standards, there is no way to access the processor's carry flag. Therefore, I simulate a carry flag. The carry flag is simulated in a C++ class, called *carried_primitive*, that provides a C++ object wrapper for type *unsigned int*. This class is derived from a class template that provides a C++ object wrapper for each numeric C++ primitive type. In order to make things look simpler when calling **`bigAdd`** from assembly code, I have given the **`bigAdd`** function a C-language interface.

It is suggested to ask on Piazza if you have any questions.

So, to summarize, you must write an assembly-language subroutine in bigAdd.S that performs arbitrary-sized unsigned addition according to these instructions.