ENSC 254 – Lab 3

Summer 2021

One student using Parallels on a Mac had to rename their .S files to .s files (or do a config. change).

1 Introduction

Up till now, we have been manually generating ARM machine code from an ARM program written in symbolic form. In this lab, we will start using a tool called an assembler to automatically generate machine code from a symbolic ARM program, often actually called an ARM Assembly-Language program.

This week, we are focusing on branching to and from subroutines, as well as using conditional branches to break out of loops and then later to avoid looping back. These are critical basic elements for programs big and small, and are how compilers for higher level programming languages like C actually implement this kind of functionality.

2 Conditional Branching and Subroutines

In the previous lab, we employed different ways to switch execution to an instruction of the program other than the usual 'next' instruction. In addition to the branch instruction (`B`), we also manipulated the program counter using the `LDR` instruction and several data processing instructions including the `MOV` instruction and, indirectly, the `SUB` instructions. In the process, we tried a couple assembly pseudo addressing modes and a pseudo instruction.

All of the methods we looked at in the last lab were unconditional branches – that is, the program always branched. It is common to make branch instructions conditional, so that the branch only occurs under a certain condition. In this lab, we examine conditional instructions (instructions that only occur when certain conditions are met), and the branch-and-link instruction, which stores a return address before branching -- allowing us to get to a so-called subroutine and then return from it.

2.1 Condition Code Flags

Many processor architectures include a *Machine Status Register* (MSR) or *Program Status Register* (PSR). ARM7 processors include such a register called the *Current Program Status Register* (CPSR). This is one of the registers that can be viewed within the Keil simulator (and it can be expanded there for more detail). Of particular interest, condition code flags in the CPSR can be set as a result of some operations. These are the ARM condition code flags:

Name	**Description**
Negative ***N***	Set equal to bit 31 of the ALU's output, and especially useful for twos complement signed values: N = 1 if the result is negative, N = 0 if the result is non-negative
Zero ***Z***	Is set to 1 if the ALU's output is 0 (can imply a comparison indicated equality).
Carry ***C***	- During addition (including the CMN instruction), 1 if the addition produces an unsigned overflow (a carry), 0 otherwise. - During subtraction (including the CMP instruction), 0 if the result is an unsigned underflow (a borrow), 1 otherwise. - During MOV and MVN instructions, C is the value of the last bit shifted out by the shifter - Otherwise, typically left unchanged
(signed) oVerflow ***V***	- For addition or subtraction, 1 if a signed overflow or signed underflow occurred (see lecture notes). - Otherwise, typically left unchanged.

Many CPU architectures use a similar set of condition code flags.

In some architectures, the PSR will always be updated as a result of operations. In others, such as the ARM7, the programmer can decide if some instructions will affect the PSR. For example, `ADD` and `ADDS` have similar behavior, except that `ADDS` will modify the CPSR and `ADD` will not.

Some instructions exist specifically to set the CPSR flags (and are often followed by conditional instructions, see Section 2.2 below). For example, the `CMP` (CoMPare) operation is equivalent to the `SUBS` operation (derived from the `SUB` operation, which directly performs subtraction), except that it does not store the ALU result into a destination register.

Some operations use the *C* condition code flag as an input to the operation. For example, ADC (ADd with Carry) uses C as a "carry in" input.

More information can be found in Section 3.3.1 (and the page before it) of *Modern Assembly Language Programing with the ARM Processor* and Sections 7.2 and 7.3 of *ARM Assembly Language (2nd Edition).*

2.2 Conditional Instructions

ARM processors support conditional execution for most instructions – the E in the most-significant nibble of the machine instructions we created in previous labs encodes unconditional "execute always". Not all processor architectures support conditional execution for non-branch instructions, but ARM does.

One of the following extensions can be used as one of possible suffixes for an instruction mnemonic. For example, to branch when the zero flag is set, the `EQ` suffix is added to `B` and the mnemonic becomes `BEQ`. An unconditional branch can be written as either `B` or `BAL` (`B` is more common). To perform addition only if the negative bit is set, you would use `ADDMI` (or `ADDMIS`, if you also want to update the status register).

Machine code [31:28]	**Mnemonic Extension(s)**	**Meaning**	**CPSR State Examined**
0000 (0x0)	EQ	Equal	Z set
0001 (0x1)	NE	Not Equal	Z clear
0010 (0x2)	CS/HS	Carry set/unsigned higher or same	C set
0011 (0x3)	CC/LO	Carry clear/ unsigned lower	C clear
0100 (0x4)	MI	Minus/ Negative	N set
0101 (0x5)	PL	Plus/ Positive or zero	N clear
0110 (0x6)	VS	Signed overflow	V set
0111 (0x7)	VC	No signed overflow	V clear
1000 (0x8)	HI	Unsigned higher	C set and Z clear
1001 (0x9)	LS	Unsigned lower or same	C clear or Z set
1010 (0xA)	GE	Signed greater or equal	N set and V set, or N clear and V clear (N == V)
1011 (0xB)	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100 (0xC)	GT	Signed greater than	Z clear, and N and V either both set or clear (Z == 0, N == V)
1101 (0xD)	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110 (0xE)	AL (or none)	Always (Unconditional)	

2.3 Branching

Branching – that is, changing the flow of execution of a program – is a pretty common thing for program code to do. Our program from last week ended up at the bottom with an unconditional branch (we tried a number of ways to achieve this):

```
        b Loop      ;@ take us back to 'ldrb r1, …'
```

An alternative for the bottom of the program is to forever loop immediately back to the branch instruction:

```
        ;@ […]

done    b done      ;@ Loop forever (a way to stop a program)
```

The assembler translates the label into a memory address, generates a relative offset from the program counter, and does other manipulations that were needed for last week's lab.

Notice that this approach can be limiting in some cases, as the instruction uses only a 24-bit signed immediate to encode all the information about the offset from the PC. As a result, it can reach approximately +/- 32 MB from the PC in the 4 GB or so memory address space of a 32-bit ARM processor.

An important variant of the instruction is *Branch and Link* (`BL`). In addition to changing the Program Counter (PC/R15), a return address to eventually be loaded back into the PC is stored in the Link Register (LR/R14). This allows a program to execute code in a subroutine, and then return to the next instruction after a BL instruction using this pattern of instructions:

```
        bl subr1           ;@ Branch execution to "subr1" and store
                           ;@    the return location in the Link Register (LR)

        <next instruct'n> ;@ Execution returns here

        ;@ ...

subr1   ;@ ...
        mov pc, lr         ;@ Return from the subroutine to <next instruct'n>
```

A related instruction for branching to the memory address stored in a register is *Branch and Exchange* (`BX`). The syntax (and an example using the Link Register) is:

```
        bx{<cond>} <Rm>       ;@ for example… bxeq lr
```

This instruction is needed when code with the 16-bit THUMB instructions are involved, but we are not using any THUMB code in this course, so will not use this instruction.

More information on branches can be found in Section 3.5 of *Modern Assembly Language Programing with the ARM Processor* and Section 8.2 of *ARM Assembly Language (2nd Edition).*

3 Fibonacci Sequence

The Fibonacci sequence is an integer sequence taught in basic math classes. The typical sequence is:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Elements of the sequence have a recursive relation that can be described as

$$F_n = F_{n-1} + F_{n-2}$$

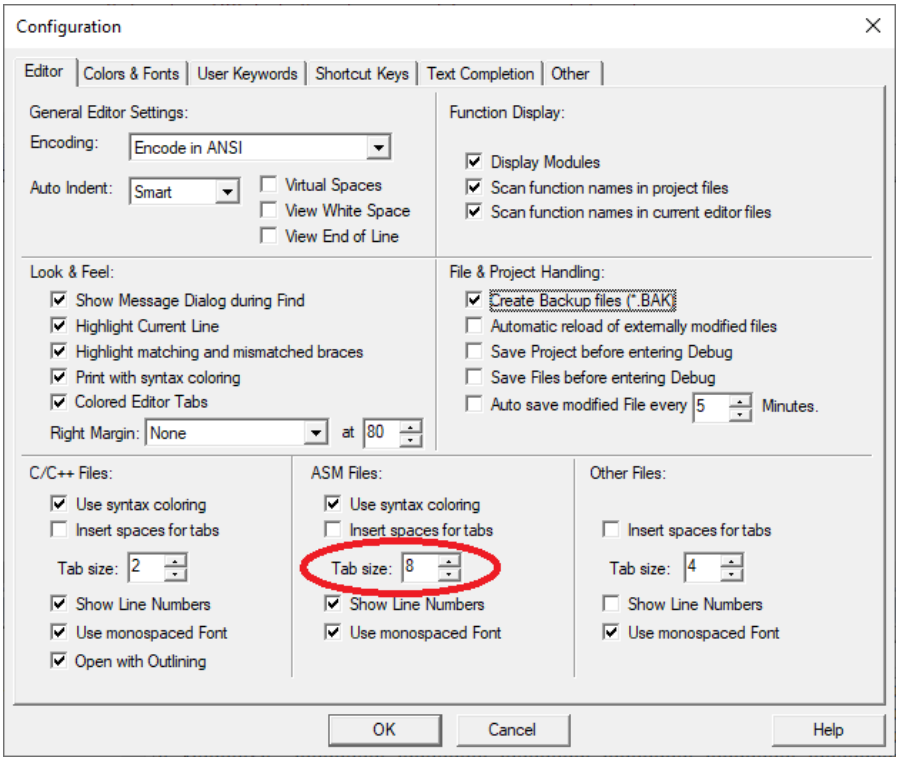
Where

$$F_0 = 0, \quad F_1 = 1$$

This sequence rapidly becomes difficult to calculate by hand, and is an example of the kind of recursive relation that can be rapidly calculated by a computer. We have distributed to you two Keil projects for developing a program to calculate elements of the Fibonacci sequence.

3.1 Simple Algorithm

We have provided a uVision project with a project file uVisionProject.uvproj in the zipped directory Lab3-64. Expand the files before opening the uVision project file. Set tabs to 8 characters for ASM files in Edit > Configuration:



We have included a simple version of an algorithm that can be used to generate the Fibonacci sequence. In this example, we use 64-bit variables and perform a 64-bit add through two 32-bit add operations. This is large enough to calculate slightly more than 90 terms of the Fibonacci sequence.

The code shown on the next page (stored in the project in directory `Lab3-64`) executes one iteration of the Fibonacci sequence – it moves the value of F_1 into variable *prev* (for previous) and then stores value F_2 into variable *curr* (for current).

Before simulating the code in the Debugger, you will have to build the project. You can use the menu item

Project > Build Target

or you can use the associated "Build" button on the screen or hit F7 on your keyboard. Now simulate the existing code.

Next you need to finish the code so that we can calculate a particular Fibonacci number, such as F_{90}. Insert a line of code where indicated to increment the register that is holding the subscript of the Fibonacci number that is now sitting in variable *curr* (so the register is holding the x in F_x). Using a branch, insert code where indicated to turn this into a loop. Then use a conditional branch where indicated near the top of the loop to exit the loop when the Fibonacci number with the desired subscript has been calculated.

You will get a warning if you try to go to the Debugger when your "executable" is outdated (you have modified the source file without building again). So make sure you build before you Debug if you have changed the source file.

```
        AREA  ||.aRESET||, CODE, READONLY, Align=3 ;@ Stored in simulated ROM
        EXPORT  done

        ;@ Pointers to the variables
        ldr r0, =prev
        ldr r1, =curr

        ;@ Load a 64-bit 0 into variable prev and 1 into curr
        mov r2, #0        ;@ Constant used for initializing the variables
        str r2, [r0, #0]  ;@ Set the value of prev
        str r2, [r0, #4]

        str r2, [r1, #0]  ;@ Set the value of curr
        mov r2, #1        ;@ Constant used for initializing LSW of curr
        str r2, [r1, #4]

        ;@ What Fibonacci subscript is variable curr holding?
        mov r5, #1

loop    cmp   r5, #90           ;@ We want to calculate F90
;@      <??????????>      ;@ *** Branch to "done" if we are finished ***

        ;@ Add the least-significant word (LSW) from each variable
        ldr   r3, [r0, #4]      ;@ Load the LSW of prev
        ldr   r12, [r1, #4]     ;@ Load the LSW of curr
        str   r12, [r0, #4]     ;@ Move the LSW of curr into the LSW of prev

        ;@ We add the two words without carry for the LSW.
        ;@ We add the other words using a carry.
        ;@ We set the status flags for subsequent operation
        adds  r3, r3, r12 ;@ Add LSWs, set status flags

        str   r3, [r1, #4]      ;@ Store LSW of result into the LSW of curr

        ;@ Add the most significant word (MSW) from each variable, with carry.
        ldr   r3, [r0, #0]      ;@ Load the MSW of prev
        ldr   r12, [r1, #0]     ;@ Load the MSW of curr
        str   r12, [r0, #0]     ;@ Move the MSW of curr into the MSW of prev

        adcs  r3, r3, r12 ;@ Add MSWs using carry bit, set status flags
        ;@ *** did it carry out (unsigned overflow)? ***

        str   r3, [r1, #0]      ;@ Store MSW of result into the MSW of curr

;@      <??????????>      ;@ *** Increment the subscript (in r5) ***
;@      <??????????>      ;@ *** Branch to "loop" ***

done    b     done        ;@ Program done! Loop forever.

;@ ********************
        AREA myData, DATA, READWRITE, Align=2
;@ variables start at address 0x4000 0000

prev    space 8                 ;@ Previous Fibonacci value (64-bit)
curr    space 8                 ;@ Current Fibonacci value (64-bit)
        END                     ;@ End of assembly in program file
```

3.2 Overflow handling and 128-bit Algorithm with Subroutines

Due to the limited size of the variables, the algorithm in the previous section could only calculate less than 100 Fibonacci numbers. In this section, you will increase the width of the addition algorithm to be able to use 128-bit (4-word, 16-byte) variables.

Furthermore, the previous approach would result in significantly larger programs as the width of the addition increases. To create a re-usable algorithm for 128-bit addition and to avoid duplication of similar code, we are going to use subroutines. (I also want to introduce subroutines at this point in the course).

Finally, we are also going to use conditional instructions to handle the normal situation (where an overall unsigned overflow did not occur) differently than when an overall unsigned overflow does occur. Since we are using unsigned integers, the overflow in question occurs when the addition of the most significant word results in the carry flag being set. The *add128* subroutine should return with carry flag cleared if there was no such unsigned overflow, and with carry flag set if an unsigned overflow did occur for the most significant word. So, after doing the 128-bit addition, we only want to increment the register holding the subscript and branch back to the top of the loop if, upon return from the *add128* subroutine, overflow was not flagged.

We have provided the uVision project file uVisionProject.uvproj in the zipped directory Lab3-128. Expand all the files before opening the uVision project file.

Now complete the program for a 128-bit implementation using subroutines.

```
;@ Tabs set to 8 characters for ASM files in Edit > Configuration

        AREA   ||.aRESET||, CODE, READONLY, Align=3
        EXPORT  overfl
        EXPORT  done

        ;@ Pointers to the variables
        ldr r0, =prev           ;@ prev is 128-bit
        ldr r1, =curr           ;@ curr is 128-bit

        ;@ Load a 128-bit 0 into variable prev and 1 into curr
        mov r2, #0        ;@ Constant used for initializing the variables
        ;@ *** Complete the initialization for prev and curr ***

        mov r2, #1        ;@ Constant used for initializing LSW of curr
        str r2, [r1, #12]

        ;@ What Fibonacci subscript is variable curr holding?
        mov r5, #1

loop    cmp    r5, #200
        beq    done

        bl     add128      ;@ Perform a 128-bit add

               ;@ *** If our variable curr did not overflow… ***
;@      <??????????>       ;@ *** Increment the subscript (in r5) ***
;@      <??????????>       ;@ ***     branch back to "loop" ***
```

```
overfl  b     overfl      ;@ Oops, the add overflowed!  Fib number in prev.

done    b     done        ;@ Program done! Loop forever. Fib number in curr.

;@ Subroutine to add 128-bit unsigned variables and move one of them.
;@      curr at r1 moved to prev at r0 and sum put in curr.
;@      Carry flag set if unsigned overflow did occur.
;@      Does not modify r0 or r1.
add128  nop               ;@ Do nothing (no operation)
;@      <??????????>

;@ We clear the carry flag to begin with.
;@ Start with the least significant word (word 0 at offset 12).
;@ We add all words using a carry.
;@ We set the status flags for subsequent operations.

        adds   r0, r0, #0  ;@ Clear the carry flag

        mov    r2, #12
        bl     doPart

        mov    r2, #8
        bl     doPart

;@ *** Complete the 128-bit addition algorithm ***

;@ *** What issue do/might we have returning from subroutine? How can we fix it?
;@      <??????????>      ;@ *** Return from subroutine ***

;@ Subroutine to load parts of operands, do a 32-bit add,
;@      move curr part into prev part and store
;@      the result of the add in place of curr part.
;@      r0 points to the beginning of prev
;@      r1 points to the beginning of curr
;@      <??> is <what ???>
;@      Does not modify r0 or r1.

;@ *** Update this subroutine to take another argument so it can
;@     be reused for processing all four words ***
doPart  ldr    r3,  [r0, <??>]   ;@ Load a value from prev
        ldr    r12, [r1, <??>]   ;@ Load a value from curr
        str    r12, [r0, <??>]   ;@    ... move into prev
        ;@ 32-bit add
        adcs   r3,  r3, r12      ;@ Add words at r3 with carry, set status flags
        str    r3,  [r1, <??>]   ;@ Store the result into curr
        mov    pc,  lr           ;@ Return from subroutine

;@ --------------------
        AREA myData, DATA, READWRITE, Align=2
;@ variables start at address 0x4000 0000

prev    space <??>        ;@ Previous Fibonacci value (128-bit) ***
curr    space <??>        ;@ Current Fibonacci value (128-bit) ***

        END               ;@ End of assembly in program file
```

Did you encountered an issue when using the Branch-and-Link instructions and returning from one of

the subroutines? How did you solve this issue?

Once the program is complete and is running, determine how many elements of the Fibonacci sequence the 128-bit design can correctly calculate. That is, what is the highest x for which we can correctly calculate F_x?