ENSC 254 – Lab 3

Summer 2019

1 Introduction

In this lab, we fully move into working in assembly language (as opposed to manually generating machine code). This week, we are focusing on branching to and from subroutines, as well as using conditional branches to control loops. These are critical basic elements to programs of any size, 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 evaluated 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 `MOV` and `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 and never came back. In this lab, we examine the conditional branch (a branch that only occurs when certain conditions are met), and branch-and-link, which stores a return address allowing us to get to a subroutine and then return from it.

2.1 Program Status Register

Many CPU architectures include a *Machine Status Register* (MSR) or *Program Status Register* (PSR). For ARM7 CPUs, this register is called the *Current Program Status Register* (CPSR). This is one of the registers that can be viewed within the Keil simulator (and can be expanded for more detail). In particular, the condition code flags are set as the result of some operations. Many CPU architectures use a set similar to these ARM flags:

Name	**Description**
Negative (N)	Set by bit 31 of the result, based on the twos complement signed value. N = 1 if the result is negative, N = 0 if the result is positive
Zero (Z)	Is set to 1 if the result was 0, or if a comparison result was equal.
Carry (C)	- During addition (including CMN), 1 if the addition produces an unsigned overflow (a carry), 0 otherwise. - During subtraction (including CMP), 0 if the result is an unsigned underflow (a borrow), 1 otherwise. - During shift operations, C is the value of the last bit shifted out by the shifter - Otherwise, typically left unchanged
Overflow (V)	- For addition or subtraction, 1 if a signed overflow occurred - For non-addition/subtraction, typically left unchanged.

In some architectures, the PSR will be always be updated as the 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 status register and `ADD` will not.

Some instructions exist specifically to set the CPSR flags for conditional instructions. For example, the `CMP` command is equivalent to the `SUBS` command, except that it does not store the result into a destination register.

More information can be found in Section 7.2 of *ARM Assembly Language (2nd Edition).*

2.2 Conditional Instructions

This section will be focusing on conditional branch instructions (a typical use of conditional instructions). The ARM7 supports conditional execution for most instructions – the E in the most-significant nibble of the machine instructions we created in previous labs encodes the unconditional "execute always". Not all processor architectures support conditional execution for non-branch instructions.

The following conditions are used as suffixes for instructions. For example, to branch when the zero bit is set, `B` becomes `BEQ`. To add a value only if the negative bit is set, you would use `ADDMI` (or `ADDMIS`, if you want to modify the status register).

Machine code [31:28]	**Mnemonic Extension**	**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 !S= 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)	

More information on conditional instructions can be found in Section 7.3 of *ARM Assembly Language (2nd Edition).*

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 example program ends with an unconditional branch, which will always execute:

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

The assembler translates the label into a memory address, generates a relative offset from the current PC value, and does other manipulations that we saw last week.

Notice that this approach can be limiting in some cases, as the instruction uses a 24-bit signed immediate. 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 related command is *Branch and Link* (BL). In addition to changing the Program Counter (PC/R15), the return address for 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 calling code using this pattern of instructions:

```
      bl subroutine     ;@ Branch execution to "subroutine" and store
                        ;@ the return location in the link register (LR)

      <next instruct'n> ;@ Execution returns here

      ;@ ...

subroutine:
      ;@ ...
      mov pc, lr        ;@ Return from the subroutine
```

A related instruction for branching to the memory address stored in a register is *Branch and Exchange* (BX). For returning from a subroutine using the link register, the syntax will be:

```
      bx lr
```

This instruction is needed when THUMB code is involved, but we are not using any THUMB code, so will not use this instruction in this course.

More information on branches can be found in 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 format 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 algorithm that can be rapidly calculated by a computer. We have distributed to you a Keil project for developing a program to calculate elements of the Fibonacci sequence.

3.1 Simple Algorithm

We have provided a uVision project in the file uVisionProject.uvproj in the zipped directory Lab3-64. Expand the files before opening the uVision project file.

Note, you might have to follow the instructions in section 3.3 to configure some per-user configuration data. Unfortunately, the Keil IDE seems not to allow us to save some settings with the project.

We have included a simplified version of the algorithm that can be used to solve 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 up to the 92nd term of the Fibonacci sequence (F_{92}).

The code shown below (stored in the project file `Lab3-64`) executes one iteration of the sequence – it stores value F_2 into var_b, and moves the value of F_1 into var_a. Using a conditional branch, insert code where indicated to turn this into a loop, calculating 90 or so more values of the sequence.

```
      .text            ;@ Store in ROM

Reset_Handler:
      .global Reset_Handler   ;@ The entry point on reset

main:
      ;@ Pointers to the variables
      ldr r0, =var_a
      ldr r1, =var_b

      ;@ Load a 64-bit 0 into variable var_a and 1 into var_b
      mov r12, #0;      @ Constant used for initializing the variables
      str r12, [r0, #0] ;@ Set the value of var_a
      str r12, [r1, #0] ;@ Set the value of var_b
      str r12, [r0, #4]

      mov r12, #1       ;@ Constant used for initializing LSW of var_b
      str r12, [r1, #4]

      ;@ Counter to specify how many terms we want to calculate
      mov r4, #90

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

      ;@ 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 operations
      adds  r3, r3, r12       ;@ Add LSWs, set status flags

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

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

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

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

;@    <??????????>            ;@ Decrement the loop counter (r4)
;@    <??????????>            ;@ Branch to "loop" if we haven't finished

done:
      b     done        ;@ Program done! Loop forever.

      .data             ;@ Store in RAM
var_a:      .space      8           ;@ Variable A (64-bit)
var_b:      .space      8           ;@ Variable B (64-bit)

      .end              ;@ End of program
```

3.2 128-bit Algorithm with Subroutines

The algorithm in the previous section could only calculate to the 92nd element, due to the limited size of the variables. In this section, you should increase the addition to use 128-bit (4-word) variables.

Furthermore, the previous approach would get larger and messier as the size of the addition increases. To help clean up the design (and allow convenient reuse code), we are going to use subroutines.

Finally, we are also going to use a conditional branch to detect if an unsigned overflow has occurred. Since we are using unsigned integers, this occurs when the addition of the most significant word results in the carry bit being set. The add_128 subroutine should return with carry flag cleared if there was no overflow, and with carry flag set if an overflow did occur.

We have provided a uVision project in the file uVisionProject.uvproj in the zipped directory Lab3-128. Expand 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

      .text                   ;@ Store in ROM

Reset_Handler:
      .global Reset_Handler   ;@ The entry point on reset

main:
      ;@ Pointers to the variables
      ldr r0, =var_a
      ldr r1, =var_b

      ;@ Load a 128-bit 0 into variable var_a and 1 into var_b
      mov r12, #0       ;@ Constant used for initializing the variables
      str r12, [r0, #0] ;@ Set a value in var_a
      str r12, [r1, #0] ;@ Set a value in var_b
      ;@ *** Complete the initialization for var_a and var_b ***

      str r12, [r0, #12]

      mov r12, #1       ;@ Constant used for initializing LSW of var_b
      str r12, [r1, #12]

      ;@ Counter to specify how many terms we want to calculate
      mov r4, #1000

loop:
      bl    add_128           ;@ Perform a 128-bit add

                              ;@ *** Detect if our variable overflowed ***
;@    <??????????>            ;@ *** If so, branch to "overflow" ***

;@    <??????????>            ;@ *** Decrement the loop counter ***
;@    <??????????>            ;@ *** Have we decremented to zero yet?***

done:
```

```
      b     done        ;@ Program done! Loop forever.

overflow:
      b     overflow    ;@ Oops, the add overflowed the variable!

;@ Subroutine to add 128-bit unsigned variables and move one of them.
;@    var_b at r1 moved to var_a at r0 and sum put in var_b.
;@    Carry flag set if overflow did occur.
;@    Does not modify r0 or r1.
add_128:
;@    <??????????>

;@ 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    processPart

      mov   r2, #8
      bl    processPart

;@ *** Complete the 128-bit add ***

;@ *** What issue do 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 var_b part into var_a part and store
;@    the result of the add in place of var_b part.
;@    r0 points to the beginning of var_a
;@    r1 points to the beginning of var_b
;@    <??> is <??? what ???>
;@    Does not modify r0 or r1.
processPart:
; @ *** Update this subroutine to take another argument so it can
; @     be reused for processing all four words ***
      ldr   r3, [r0, <??>]    ;@ Load a value from var_a
      ldr   r12, [r1, <??>]   ;@ Load a value from var_b
      str   r12, [r0, <??>]   ;@    ... move into var_a
      ;@ 32-bit add
      adcs  r3, r3, r12       ;@ Add words at r3 with carry, set status flags
      str   r3, [r1, <??>]    ;@ Store the result into var_b
      mov   pc, lr            ;@ Return from subroutine

      .data                   ;@ Store in RAM
var_a:      .space <??>       ;@ Variable A (128-bit) ***
var_b:      .space <??>       ;@ Variable B (128-bit) ***

      .end              ;@ End of program
```

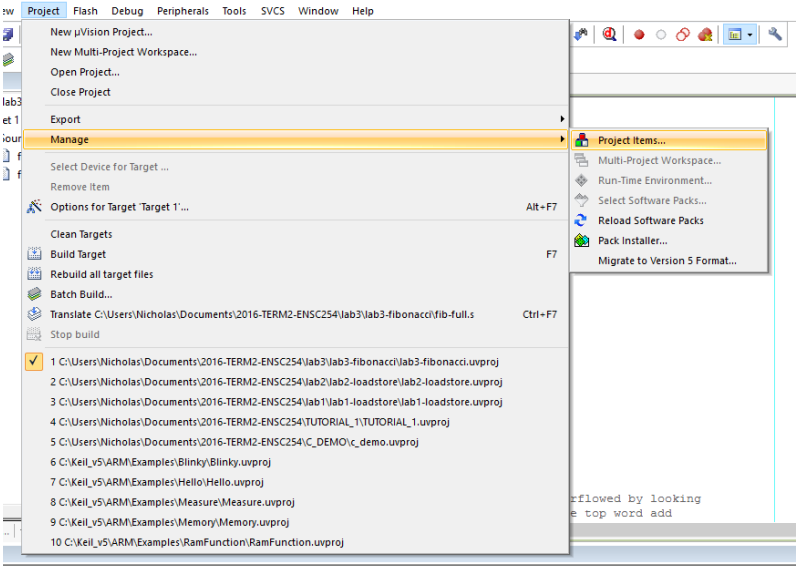
What was the issue you encountered with using the Branch and Link command and returning from subroutines? How did you solve this issue?

Once the program is complete and is running, determine how many elements of the Fibonacci sequence

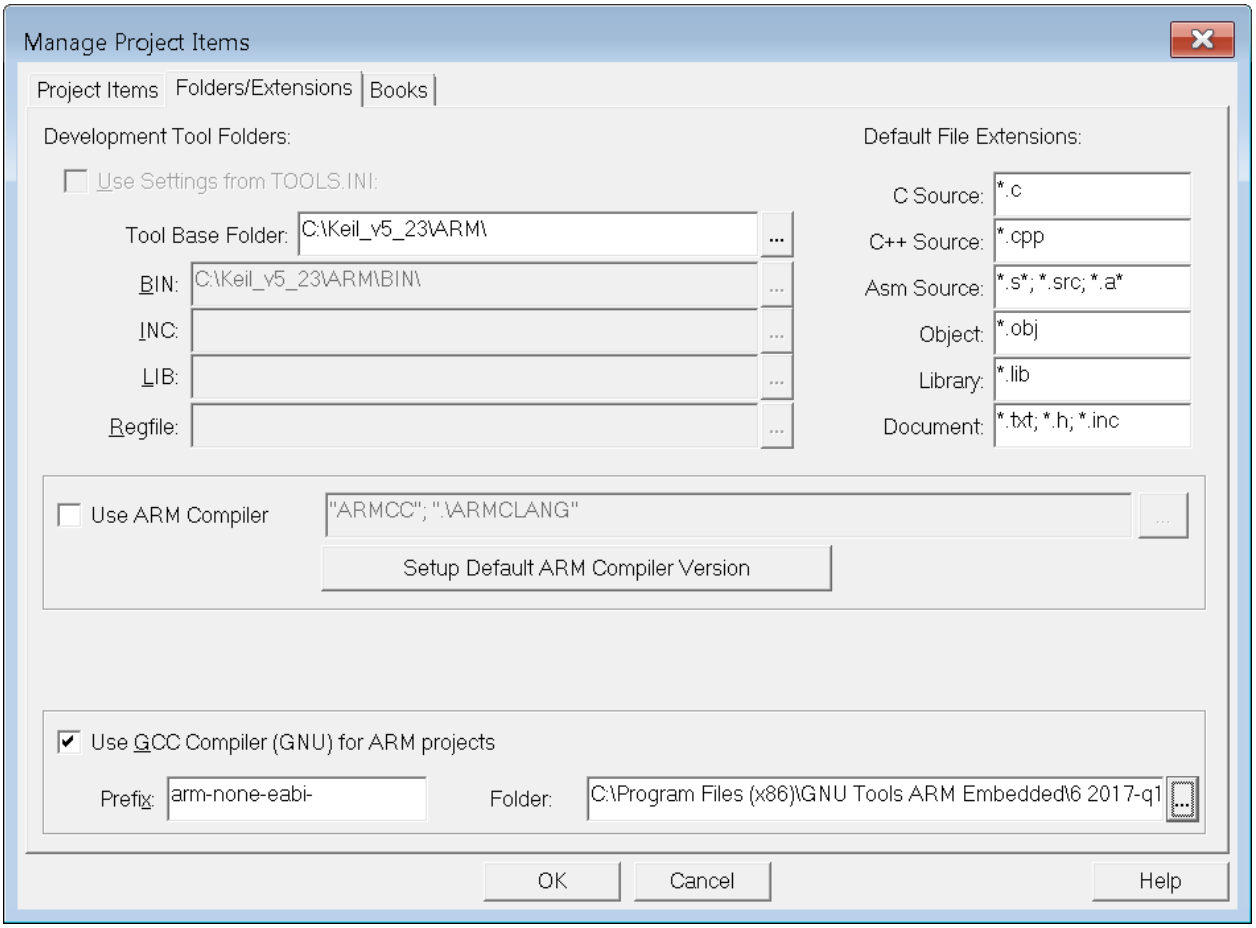
the 128-bit design can calculate.

3.3 Configuring the Build Environment

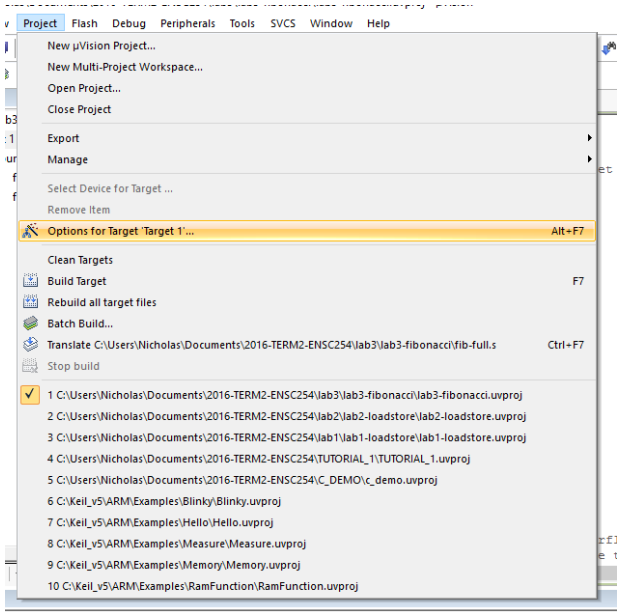
While the included project should be properly configured by default, it's possible that your system is not properly set up



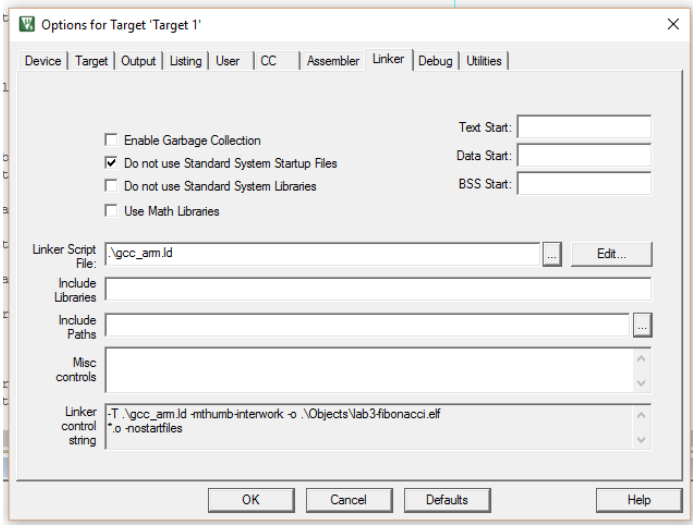
First, make sure the GCC toolchain is enabled by going to *Project → Manage → Project Items* Ensure "*Use GCC Compiler*" is enabled. The lab computers are configured with a version or two of the toolchain and they are typically installed in *C:\Program Files (x86)\GNU Tools ARM Embedded* Especially on your own computer, you might need to make sure that the Folder for GCC is indicating the correct folder for your version of GCC. The folder should be correct in the lab. You may need to exit uVision and then re-open the project after changing this setting.



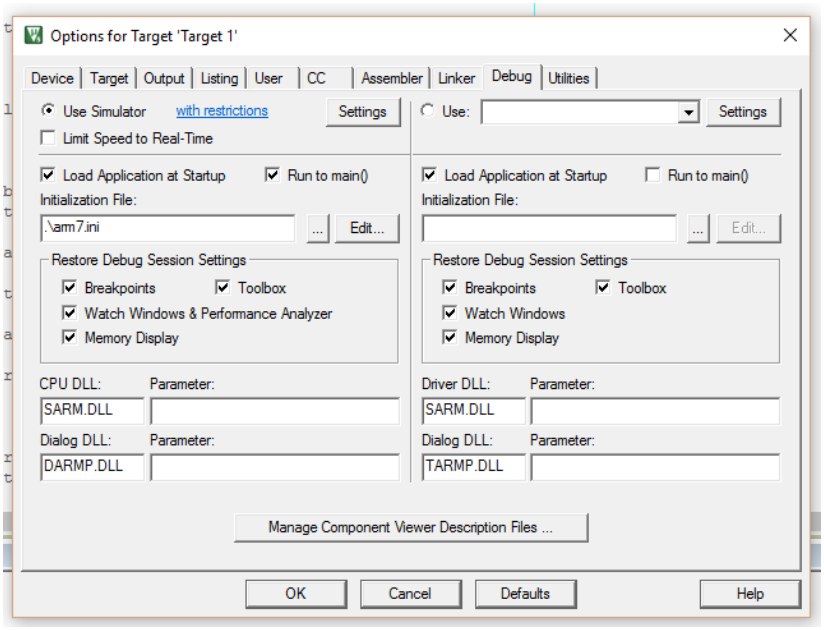
To properly use the GCC toolchain and the debugger tool, our project includes two auxiliary files that need to be included. The project options can be selected as follows:



As we discussed last week, the GNU Assembler makes use of a linker script (`gcc_arm.ld`). Ensure that it is being used (it is included as part of the project).



We also use an initialization script to configure the simulator (`arm7.ini`). Make sure it is being used in the Debug settings.



If you haven't already done so, you should probably exit uVision and then re-open the project in the program.