# *Carleton University*
# *Department of Systems and Computer Engineering*
# *SYSC 3006 Fall 2016*
# *Computer Organization*
# *Lab #7*

**Prelab Due:** Your Prelab should be done *before* the start of your scheduled lab session. Bring your Prelab with you in hard or soft copy form (e.g. print it or have a word/pdf document ready) for TA inspection during the lab session (no online submission).

**In this lab you will:**

- Program a given computer system.
- Implement and test using the Assembler and Debugger (Logisim) circuit.

**Read this document carefully before deciding what to do.**

The Software Tools needed for this lab are posted on culearn under the course Resources section. The tools include the Assembler editor and the Debugger circuit. The Debugger implements a computer system with the instruction set described in lectures. The Debugger also provides an interface for controlling software execution. The Assembler assembles source code for the instruction syntax discussed in class into a memory image (called Object Code) that can be loaded into the Debugger's Main Memory component.

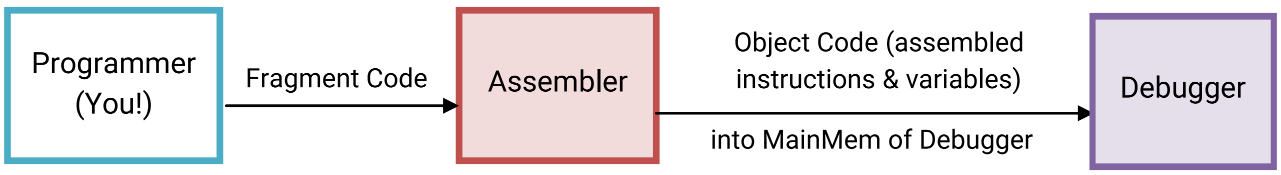

**Figure 1.** Overview of the software development and debugging (testing) workflow in this lab.

## Fragment 1

The Fragment1_SRC.txt file contains the following assembly language source code fragment:

```
    B   SkipOverVariables

 Arr_Size  DCD   #3   ; Arr is an array of 3 words
 Arr
    DCD  #20     ; first (0-th) element of Arr = 20
    DCD  #-4     ; second (1-th) element of Arr = -4
    DCD  #0      ; third (2-th) element of Arr = 0

 SkipOverVariables
    MOV  R2, Arr
    LDR  R3, [ Arr_Size ]
    CMP  R3, #0
    BEQ  Done
    SUB  R3, R3, #1

 Loop
    LDR  R5, [R2, R3 ]
    ADD  R5, R5, #10
    STR  R5, [R2, R3]
    SUB  R3, R3, #1
    BPL  Loop

 Done
    DCD  #0xFFFFFFFF    ; breakpoint instruction
```

Be sure to see the note about the DCD directive at the end of this lab document. DCD was discussed briefly in lectures, but here the syntax is described in detail.

**Prelab Questions about Fragment 1**:

PQ1: What is the high-level objective (purpose) of the code fragment? Explain, in a single sentence, the objective in terms of the net effect of the fragment on the variables it modifies.

PQ2: Write C-like pseudocode that accomplishes the same objective (see below for an example of C-like pseudocode).

PQ3: Starting after the SkipOverVariables declaration, add comments to the instructions that document what is being done ... the comments should be at the level of the pseudocode objective, not at the RTL level. For example, consider the instruction:

```
MOV  R4, #1
```

An RTL-level comment for the instruction might be: "; move #1 into R4", which is accurate but says nothing about the net programming objective (i.e. why is loading #1 useful in the context of the program's objective?). A more appropriate comment might be: " ; R4 = address of Arr_Size".

PQ4: When the fragment is executed, how many instructions will be executed (including the breakpoint instruction)?

PQ5: When assembled, how many words of memory will the fragment occupy?

**Tutorial**: Assembling and debugging Fragment 1. [Note: If you already know how to use the Assembler and the Debugger, skip this section and jump directly to Fragment 2]

To understand how to assemble and debug a given fragment, follow this video tutorial (5 minutes; no audio) which uses Fragment 1 as an example. Ensure that by the end of the tutorial, you have similar main memory values (Fragment1_MMem.txt) . The video is sufficient for the purposes of this lab. For more details, you can always refer to the pdf guides accompanying each tool on culearn: Debugger User Guide and Assembler User Guide.

## Fragment 2

Consider the following C-like pseudocode[1] that processes an array to replace every element with its absolute value:

```
for ( R11 = 0; R11< Arr_Size; R11++ )    ; we'll choose R11 to store array index, start with R11=0
{
   if ( Arr[ R11 ] < 0 ) {
       Arr[ R11 ] = abs( Arr[ R11 ] )     ; abs() is absolute value function
    }
 }
```

A template for the assembly language code to implement this pseudocode is provided in Fragment2_SRC.txt. In the SRC file, the C-like pseudocode is embedded as comments (along with some additional comments) to help guide your code development.

In the lab: complete the code by replacing all occurrences of "***" with the necessary details and execute the fragment for the data values in the template. Do not add additional instructions.

Demonstrate your work to the TA by preparing the following 3 files:

1. **Fragment source code:** save your completed source code as "Fragment2_SRC.txt"
2. **Assembled object code:** after assembly, save your object code as "Fragment2_OBJ.txt"
3. **Execution results**: after debugging the entire program, save your Debugger Main Mem contents (right-click RAM -> Save Image) as "Fragment2_MMem.txt"

[1] Pseudocode is code that is concise, but does not necessarily have a compiler. In this case (as in class), the pseudocode refers to registers instead of variables. This pseudo code also uses assembly language-style comments (;) instead of C-style comments (//).

## Fragment 3 [OPTIONAL]

The following C-like pseudocode sorts an array of integers in ascending order:

```
for ( i = 0; i < Arr_size; i++ )
{  ; swap Arr[ i ] with smallest remaining in unsorted portion of array
   tempSmall = Arr[ i ]  ; initially assume first element is smallest remaining
   for ( j = i + 1; j  < Arr_size; j++ )
   {  if ( Arr[ j ] <  tempSmall )
      {   ; swap  Arr[ i ] and Arr[ j ]!
          Arr[ i ] =  Arr[ j ]
          Arr[ j ] = tempSmall
          tempSmall = Arr[ i ]
      }
   }
}
```

An assembly language template has been given in Fragment3_SRC.txt.

Complete the code by replacing all occurrences of "***" with the necessary details and execute (i.e. assemble and debug) the sort for the data values in the template. Do not add additional instructions.

To demonstrate your work to the TA, save the fragment SRC, OBJ, and MMem files (as done in Fragment 2).

Good Luck! ☺

## DCD Directive

The DCD (DeClare Data) directive is used to declare a memory location to be used as a variable. The general syntax for the directive is:

[Label] DCD [numeric-literal] [; comment]

Anything enclosed in [ and ] is optional. If present, the Label is the name of the variable. If present the numeric-literal defines the initial value that should be loaded into the variable's memory location when the program is loaded. As always, comments are optional.

Here are some examples:

```
        DCD             ; reserves an un-named, un-initialized word

X       DCD             ; reserves a memory word for an un-initialized variable named X

Y       DCD     #12     ; reserves a memory word for a variable named Y, initialized to 12 (decimal)

X       DCD     #0x10   ; reserves a memory word for a variable named Z, initialized to 10 (hex)
```

Note that decimal values for DCD numeric-literals may be specified as negative (using 2's complement encoding). For example

```
Minus1  DCD     #-1     ; reserves a memory word for a variable named Minus1, initialized to -1
                        ; the variable is initialized to the value 0xFFFFFFFF i.e. -1 under 2's complement
```