Name: Date:

Lab group: Email:

LABORATORY MANUAL

CE1106/CZ1106
Computer Organization and Architecture

Lab Experiment #1

Basic Assembly Language Programming using the VisUAL ARM emulator

by
A/P Goh Wooi Boon

SESSION 2020/2021
SEMESTER 2

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY

1. OBJECTIVES

1.1 Understand the steps required to assemble and execute an assembly program in the VisUAL ARM emulator and be able to interpret the observed changes in the visual environment during program execution.

1.2 Understand the characteristics of the registers and memory in the ARM processor.

1.3 Understand the characteristics and behavior of different addressing modes and be able to write a simple assembly program to access and process data in memory.

2. LABORATORY

This experiment is conducted at the **Hardware Lab 2** at **N4-01b-05 (Tel: 67905036)**.

3. EQUIPMENT

3.1 Hardware

- Personal Computer that is compatible with the Java 8 Runtime Environment.

3.2 Software / User Manuals

- The VisUAL User Guide
  - https://salmanarif.bitbucket.io/visual/user_guide/index.html
- The ARM instruction set subset supported by VisUAL
  - https://salmanarif.bitbucket.io/visual/supported_instructions.html
- The VisUAL ARM emulator software Version 1.27 (Release 29/12/2015)
  - http://bit.ly/visualwin_127 (Windows (64-bit) version)
  - https://salmanarif.bitbucket.io/visual/downloads.html (For other versions)

4. INTRODUCTION

The basic function of a microprocessor is to execute programs, and it requires both memory and internal registers to achieve this objective. Memory is usually available in large amounts to provide storage for both programs and the data, while registers are available in small quantity to provide temporary storage for data that are currently being processed or will be processed soon.

Assembly programs are normally developed using software tools that allow the program to be edited, assembled, executed and debugged. Such tools usually provide a simulated environment where program instructions can be executed and their effects observed without the presence of the physical processor. In this experiment, the VisUAL emulator software will be used to study the characteristics of registers, memory and instruction set associated with the ARM processor. Unified Assembler Language (UAL) is a common syntax for ARM and Thumb instructions. Code written using UAL can be assembled for any ARM processor. You will execute some simple pre-written ARM assembly language program and in so doing, you will develop a better understanding of the following issues:

(1) The characteristics and functions of various registers in the ARM processor.
(2) How data and programs are stored in memory.
(3) How data can be moved between memory and registers using various addressing modes.
(4) How different ARM mnemonics behave and their influence of different registers and memory.
(5) How improvements in execution performance (measured by cycles) can be obtained by using appropriate addressing modes to implement the same functions.

4.1 Notations used in this manual

`0x1234ABCD` is used to specify a 32-bit hexadecimal (hex) value of $1234ABCD_{16}$.

5. EXPERIMENT

5.1 The VisUAL ARM emulator

The VisUAL ARM emulator software provides a graphical user interface that allows you to load or create ARM assembly programs for execution, including single step execution. Figure 1 shows a screen-shot of a typical VisUAL emulator display.

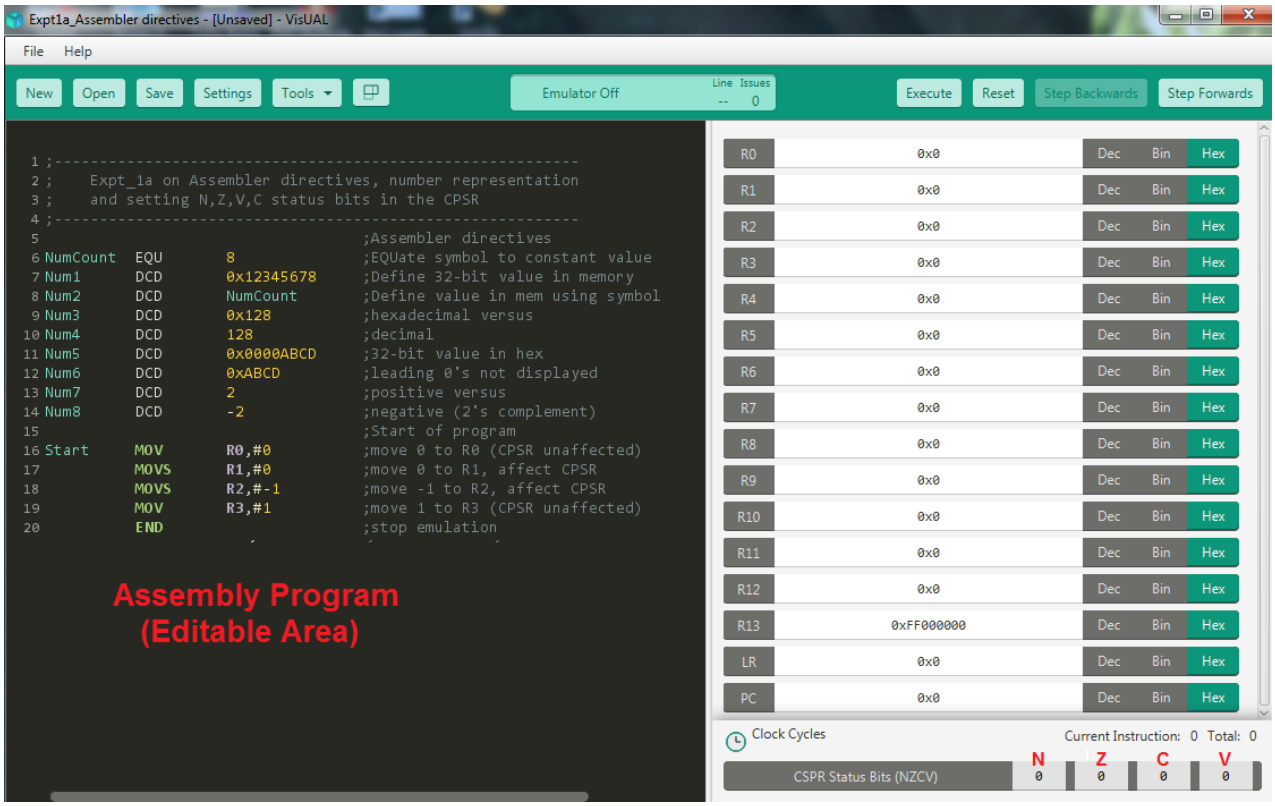


Figure 1 – Menu items, user input regions, simulation control panel and views in VisUAL

5.1.1 First run up the ARM emulator program by executing the program **VisUAL.exe** on your PC.

5.1.2 Several pre-written ARM assembly language programs have been written for lab experiment #1. To access them, follow the instructions in the ***CX1106_Expt_1_Readme*** file in the **CX1106** folder on the PC Desktop.

Remember: Since you are working on a Virtual Desktop in the lab's PC, you are advised to back up your files into your thumb drive or network drive before you logoff.

5.2 The memory and data representation

The memory is an integral part of a microprocessor system. In this section, you will study the characteristics of the memory associated with the ARM processor and the different ways numerical data is represented in memory. You will also learn how the **DCD** assembler directive can be used to initialize memory contents with specific values.

5.2.1 **Load assembly program** – Using the **Open** command, load in the ARM assembly language program ***Expt_1a_Assembler_directives_and_CPSR*** shown in Figure 2 into VisUAL.

```
1           ;-------------------------------------------------------
2           ;         Expt_1a on Assembler directives, number representation
3           ;         and setting N,Z,V,C status bits in the CPSR
4           ;-------------------------------------------------------
5           ;Assembler directives
6  NumCount EQU       8              ;EQUate symbol to constant value
7  Num1     DCD       0x12345678     ;Define 32-bit value in memory
8  Num2     DCD       NumCount       ;Define value in mem using symbol
9  Num3     DCD       0x128          ;hexadecimal versus
10 Num4     DCD       128            ;decimal
11 Num5     DCD       0x0000ABCD     ;32-bit value in hex
12 Num6     DCD       0xABCD         ;leading 0's not displayed
13 Num7     DCD       2              ;positive versus
14 Num8     DCD       -2             ;negative (2's complement)
15          ;Start of program
16 Start    MOV       R0,#0          ;move 0 to R0 (CPSR unaffected)
17          MOVS      R1,#0          ;move 0 to R1, affect CPSR
18          MOVS      R2,#-1         ;move -1 to R2, affect CPSR
19          MOV       R3,#1          ;move 1 to R3 (CPSR unaffected)
20          MOVS      R4,#0x80000000 ;move 0x80000000 into R4
21          ADDS      R5,R2,R3       ;add 1 and -1
22          ADDS      R6,R4,R4       ;add 0x80000000 and 0x80000000
23          END                      ;stop emulation
```

Figure 2 – Listing of the *Expt_1a_Assembler_directives_and_CPSR* ARM assembly program

5.2.2 **Viewing memory contents** – Open the **View memory contents** window using the **Tools** pull-down menu. Then run up the sequence of assembler directives shown in Figure 2 by clicking on **Step Forwards**. This also executes the first ARM instruction at line 16.

View Memory Contents

Start address: 0x100

Addresses of each byte in 1st Word (0x12345678)

Word Address	0x103 Byte 3	0x102 Byte 2	0x101 Byte 1	0x100 Byte 0	Word Value
0x100	0x12	0x34	0x56	0x78	0x12345678
0x104					
0x108					
0x10C					
0x110					
0x114					
0x118					
0x11C					

Table 1 – The contents in memory allocated by the various DCD assembler directives in *Expt_1a_Assembler_directives_and_CPSR* program (to be completed). Note: The default start address of the Data Memory in the VisUAL simulator is at hexadecimal **`0x0100`**. Ensure start address in the **Tools/ View memory contents** window is set to **`0x100`**.

5.2.3 **Memory Contents** – Using Table 1, fill in the hexadecimal values in memory for variables **`Num1`** to **`Num8`** allocated by the eight **`DCD`** assembler directives shown in Figure 2.

5.2.4 **Byte Ordering** – The memory variable **`Num1`** has been allocated a 4-byte (32-bit) numerical value of hexadecimal **`0x12345678`**. Given that the highest address in the 4-byte word is Byte 3 and the lowest is Byte 0. Which byte-ordering format does VisUAL adopt?

◯ **Big Endian** or ◯ **Little Endian**

5.2.5 **The EQU Assembler Directive** – Give the value of the 4-byte word starting at address **`0x104`**? Which assembler directive allocated this value into the memory?

Give the value of the symbol labelled **`NumCount`**. How was it assigned with this value?

5.2.6 **Hexadecimal versus Decimal** – By observing the two **`DCD`** directives in lines 9 and 10 of the assembly program, explain why the 4-byte words at addresses **`0x108`** (**`Num3`**) and **`0x10C`** (**`Num4`**) are different?

Complete the following hexadecimal and decimal conversions in the Table 2. You can do this by switching between **Dec** and **Hex** Word Value Format in the **View Memory Contents** window or by making appropriate changes to the values in the **`DCD`** directives in lines 9 and 10. Do remember to click on **Reset** to stop code execution before you edit your assembly program. Click on **Step Forwards** again to see results of your changes.

Hexadecimal	Equivalent Decimal Value
`0x10`	
	`10`
	`255`
`0x12345678`	

Table 2 – Hexadecimal – Decimal equivalent values

5.2.7 **Leading Zeroes** – Observe the two **`DCD`** directives in lines 11 and 12 of the assembly program in Figure 2. Do both directives give the same 4-byte words at addresses **`0x110`** (**`Num5`**) and **`0x114`** (**`Num6`**)? What can you say about the way the VisUAL assembler treats and display leading zeroes in hexadecimal number notation?

State how many bytes in memory will be occupied by the assembler directive:

```
Num9    DCD    0x1
```

5.2.8 **2's Complement Representation** – Observe the two **`DCD`** directives in lines 13 and 14 of the assembly program. What are the 32-bit hexadecimal values allocated for the decimal values of **`2`** and **`-2`** at addresses **`0x118`** (**`Num7`**) and **`0x11C`** (**`Num8`**) respectively?

Value at **`0x118`** (**`Num7`**) Value at **`0x11C`** (**`Num8`**)

Give the 32-bit hexadecimal values for the negative decimal number in Table 3.

Decimal Values	Hexadecimal Equivalent (32-bit)
`-1`	
`-3`	

Table 3 – 32-bit hexadecimal values of negative decimal numbers in 2's complement

5.3 Status bits in the Current Program Status Register (CPSR)

The VisUAL emulator displays the current state of the four status bits (**N**, **Z**, **V**, **C**) in the CPSR at the bottom right corner of the VisUAL window. The interpretation of the status bits are as follows; **N**-Negative, **Z**-Zero, **V**-Overflow and **C**-Carry.

5.3.1 **MOV versus MOVS** – Using the same *Expt_1a_Assembler_directives_and_CPSR* assembly program loaded earlier, re-start the execution of the program by clicking on **Reset** followed by **Step Forwards**. Note: The highlighted instruction is the one that has just been executed.

a) What 32-bit value was loaded into the register **R0** by the execution of the `MOV R0,#0` instruction? Did this set the **Z** (Zero) flag?

b) Now execute the next instruction by clicking **Step Forwards** again. What 32-bit value was loaded into the register **R1** by the execution of the `MOVS R1,#0` instruction? Did this set the **Z** (Zero) flag? What can you say about adding the {`S`} set bit option to the `MOV` mnemonic?

5.3.2 **The N (Negative) flag** – Step through instruction `MOVS R2,#-1` at line 18. Which CPSR status bit was influenced by executing this instruction and why was the status bit affected?

Now execute the next instruction `MOV R3,#1`. What do you observed about the **N** flag? Do you expect it to be cleared, set or unaffected? Explore changing `MOV R3,#1` to `MOVS R3,#1` and see what happens to the **N** flag under this circumstances.

Next step through the `MOVS R4,#0x80000000` instruction at line 20. What do you observe about the **N** flag in the CPSR? Why do you think the **N** flag was set even when there was no negative (-ve) sign in front of the immediate data value (e.g. `-1`).

Which of the four status bits (`V`, `N`, `Z`, `C`) can be affected by the `MOVS` instruction? Can you think of an example of a mnemonic that can potentially set all the four status bits?

5.3.3 **Addition and the CPSR status flags** – Table 4 shows examples of the addition of different 32-bit values and how they influence the V, N, Z, C flags. First, manually solve the addition problems given below and predict what would be the value (i.e. 0 or 1) of the V, N, Z, C flags and 32-bit results. You can now step through the instructions at lines 21 and 22 to compute the addition of the two examples given in Table 4. Check if your setting of the CPSR status flags tallies with that given by the ARM simulator.

Signed overflow is indicated by the setting of the **V** flag and unsigned overflow is indicated by the setting of the **C** flag. Based on the signed and unsigned interpretation of the hexadecimal values in Table 4, determine if signed and/or unsigned overflow has occurred in each of the addition. Confirm if the respective setting of the **V** and/or **C** flags tallies with what you worked out.

0x00000001	CPSR Flags				**0x80000000**	CPSR Flags			
+**0xFFFFFFFF**	**V**	**N**	**Z**	**C**	+**0x80000000**	**V**	**N**	**Z**	**C**
	☐	☐	☐	☐		☐	☐	☐	☐
Signed overflow: ○ Yes or ○ No					Signed overflow: ○ Yes or ○ No				
Unsigned overflow: ○ Yes or ○ No					Unsigned overflow: ○ Yes or ○ No				

Table 4 – Some addition examples and their influence on the N, Z, V, C flags

5.4 Addressing Modes and the Load/Store Instructions

Open the ARM assembly language program ***Expt_1b_Addressing_modes*** shown in Figure 3 in your VisUAL ARM emulator. This program allows you to explore the various ARM addressing modes and their effects on registers and memory locations. You will also understand how the Load (**LDR**) and Store (**STR**) instructions allow you to access content in memory.

5.4.1 **Addressing Modes** – Open the **View memory contents** window if it is not already opened. Step through each instructions from lines 11 to 18. Carefully observe the values in the registers listed in each of the instruction so that you can fill in columns 3, 4 and 5 in Table 4.

5.4.2 **Identifying the addressing modes** – Based on the observations of your entries in Table 4, identify the addressing modes of each source operand from lines 11 to 18 and describe how each addressing mode produced the observed values in their respective destination registers.

Note: The **View memory contents** window allows you to be aware of what 32-bit hexadecimal values are in addresses **0x100**, **0x104** and **0x108**.

5.4.3 **Effects on the pointer register contents** – Instructions in lines 15 to 18 use register **R2** as a pointer register to retrieve data from memory. Observe how the different addressing modes changes the address value in register **R2**. Firstly, which of these four addressing modes modifies the content in **R2** after execution?

Secondly, for each of the two source addressing modes in lines 17 to 18, state when the offset value 4 is added to the contents in pointer register **R2**, before or after the computation of the effective address. Note: The effective address is the address from where the memory content was accessed during data transfer.

```
1           ;-------------------------------------------------------
2           ;Expt_1b  - ARM addressing modes, load & store instructions
3           ;-------------------------------------------------------
4           ;Allocate values in data memory
5  DataArea EQU      0x00000100     ;data area starts at 0x0100
6  Num1     DCD      0x1111AAAA     ;allocate 32-bit values
7  Num2     DCD      0x2222BBBB     ;in memory starting at
8  Num3     DCD      0x3333CCCC     ;address 0x00000100
9  Num4     DCD      0x4444DDDD     ;
10          ;Start   of program
11 Start    MOV      R0,#0xAB       ;immediate data
12          MOV      R1,R0          ;register direct
13          ;load    register instructions
14          MOV      R2,#DataArea   ;initialize address pointer R2
15          LDR      R3,[R2]        ;register indirect
16          LDR      R4,[R2,#4]     ;base plus offset
17          LDR      R5,[R2],#4     ;auto-index (post-index)
18          LDR      R6,[R2,#4]!    ;auto-index (pre-index)
19          ;store   register instructions
20          MOV      R2,#DataArea   ;initialize R2 to start of data area
21          STR      R0,[R2]        ;store R0 to address 0x0100
22                                  ;store R0 to address 0x010C
23                                  ;store R0 to 0x0104 & increment R2 by 4
24                                  ;get the value of zero into R0
25                                  ;store R0 to 0x0104 & increment R2 by 4 again
26          END
```

Figure 3 – Listing of the *Expt_1b_Addressing_modes* ARM assembly program

1	2	3	4	5	6
Source Line No.	**Mnemonic**	**32-bit value in destination register**	**Source of 32-bit content**	**Which register(s) were modified after execution**	**Clock Cycles**
11	**MOV R0,#0xAB**	**R0=0x000000AB**	Immediate Data **#0xAB**	Only **R0**	1
12	**MOV R1,R0**	**R1=**			
14	**MOV R2,#DataArea**	**R2=**			
15	**LDR R3,[R2]**	**R3=**	Memory Address **0x00000100**		
16	**LDR R4,[R2,#4]**	**R4=**			
17	**LDR R5,[R2],#4**	**R5=**			
18	**LDR R6,[R2,#4]!**	**R6=**			

Table 5 – ARM addressing modes, their characteristics and the **LDR** instruction.

5.4.4 **Clock Cycles** – The number of clock cycles taken by the current instruction just executed is shown on the right side of **Current Instruction** (see bottom right hand corner of the VisUAL window). **Reset** your program and step through each instruction from lines 11 to 18 again. Note the number of clock cycles taken by each instruction in column 6 in Table 5.

a) The clock cycle counts taken by the **MOV** and **LDR** instructions are related to the number of memory accesses taken to fetch and execute the instruction. Why do you think **MOV R0,#0xAB** takes only 1 clock cycle while **LDR R3,[R2]** takes 2 clock cycles?

5.4.5 **Applying addressing modes to the STR instruction** – The Store (**STR**) instructions allows you to copy the 32-bit contents in a source register into a specific memory location. Using what you have learnt about the various ARM addressing modes, fill in the missing information in columns 2 and 4 in Table 6 based on the given comments shown in Figure 4.

Edit the ***Expt_1b_Addressing_modes*** assembly program in Figure 3 with your proposed mnemonics. Then step through each instruction and confirm that the addressing modes you have used with your **STR** instructions produces the expected results.

```
19  ;store register instructions
20  MOV       R2,#DataArea    ;initialize R2 to start of data area
21  STR       R0,[R2]         ;store R0 to address 0x0100
22                            ;store R0 to address 0x010C
23                            ;store R0 to 0x0104 & increment R2 by 4
24                            ;get the value of zero into R0
25                            ;store R0 to 0x0104 & increment R2 by 4 again
26  END
```

Figure 4 – Listing of the **STR** instructions in the *Expt_1b_Addressing_modes* ARM assembly program.

1	2	3	4	5
Source Line No.	**Mnemonic**	**32-bit value in destination after execution**	**Source of 32-bit content**	**Which register(s) were modified after execution**
20	**MOV R2,#DataArea**	**R2=0x00000100**	Immediate Data **#0x00000100**	Only **R2**
21	**STR R0,[R2]**	**0x100=0x000000AB**	Register **R0** **0x000000AB**	None
22	**STR R0,**	**0x10C=0x000000AB**	Register **R0** **0x000000AB**	None
23		**0x104=0x000000AB**	Register **R0** **0x000000AB**	Only **R2**
24		**R0=0x00000000**		Only **R0**
25		**0x104=0x00000000**	Register **R0** **0x00000000**	Only **R2**

Table 6 – ARM addressing modes and the **STR** instruction.

Applications #1 – The incomplete ARM assembly program in Figure 6 initializes an integer array of 4 elements labeled **Num** with its first element assigned the value given by **FirstN** and then each subsequent array element assigned a value incremented by the amount **Steps**. The array **Num** starts at address **0x0108**. Constant values **FirstN** and **Steps** are stored as 32-bit values in memory addresses **0x0100** and **0x0104** respectively.

Write an ARM assembly program to achieve the stated requirements. You are free to use the suggested comments in the listing give in Figure 6 or write your own version of the program that can achieve the same objective. Type out your completed assembly program using the incomplete VisUAL code file ***Expt_1c_Application_1***. Open the **View memory contents** window before you step through the program and observe that it indeed produces the result you expect based on the given values in **FirstN** and **Steps**. If your program executes correctly, you may see the memory map shown in Figure 5 (but 0x118 may not appear).

Hint: You can write an efficient code that is both short in length and executes optimally with minimum clock cycles if you can employ the most efficient addressing mode that can allow you to load up memory constants into appropriate registers and index through the array **Num** while storing the incrementing value in register **R0**.

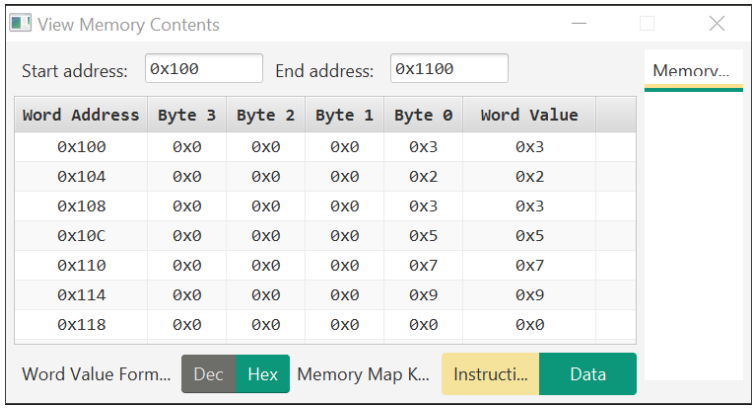


Figure 5 – The expected memory map after the correct execution of *Expt_1c_Application_1* code

```
;-------------------------------------------------------------------
;Expt_1c - Application #1 - Array of incrementing numbers
;-------------------------------------------------------------------
DataArea    EQU   0x00000100       ;data area starts at 0x0100
ASize       EQU   4                ;Array size
FirstN      DCD   3                ;value of the first number in array
Step        DCD   2                ;increment steps
Num         FILL  ASize            ;allocate memory for 4 integers
            ;Start of program
Start       MOV   R3,#DataArea     ;initialize pointer R3 to start of data area
            MOV   R2,#ASize        ;initialize array size into counter R2
                                   ;load memory constant FirstN into R0
                                   ;load memory constant Step into R1
Loop                               ;store R0 into current array element
                                   ;increment R0 by Step
                                   ;decrement counter R2 by 1
            BNE   Loop             ;loop back if R2 is still not zero
            END
```

Figure 6 – Listing of the incomplete *Expt_1c_Application_1* ARM assembly program

Note: The remaining sections are optional. You should proceed to complete it if you have already completed section 5.1 to 5.4.

5.5 Optimising an ARM assembly program

There are two different objectives one can use to optimise an assembly program. You can re-write the original assembly code to provide the same functionality but now using fewer number of instructions (optimise for size) or executes using fewer clock cycles (optimise for speed). It is not uncommon (but not always so) that optimised codes that execute faster are also generally smaller in size, and vice versa.

Now that you have executed a simple assembly language program, re-write the ARM assembly program shown in Figure 7 so that you can count the length of the string in memory (starting at the label **`String_A`**) using as few clock cycles as possible. The length of the string in number of bytes (excluding the string terminator **`Null_char`**) should be placed in the memory variable **`Count_A`** at the end of the program execution, which should have the value **`0x0000000C`** (i.e. 12 bytes).

Some hints to help you optimize your code:

1. Focus on reducing the number of cycles of the repeatedly executed code segment within the loop structure as this provide significant execution speed improvement to the overall program.
2. Consider the use of some of the addressing modes (e.g. auto-indexing) you have learnt in section 5.4 to reduce the number of instructions needed when accessing memory.
3. Reduce the number of times you access memory by using temporary registers where possible.

Note: The program in Figure 7 requires 167 cycles to compute the length of the string **`String_A`**. It is possible to optimise the code to do the same using only 83 cycles. Any reduction is cycle count is still considered optimisation. See if you can at least achieve 107 cycles.

```
1           ;-------------------------------------------------------
2           ;Expt_1d  Application #2 - String Length Counting
3           ;1.       Count the length of the string starting at label String_A
4           ;2.       Memory variable Count_A contains the length after execution
5           ;Speed:   167 cycles
6           ;-------------------------------------------------------
7           ;Allocate variables and constants into data memory
8  DataArea EQU       0x00000100     ;data area starts at 0x0100
9  Null_char EQU      0              ;declare value of null character
10 Count_A  FILL      4              ;create int memory variable for String length
11 String_A DCD       0x6C6C6548     ;place the string "Hello World!"
12 Str_A1   DCD       0x6F57206F     ;into memory using their
13 Str_A2   DCD       0x21646C72     ;individual ASCII characters
14 Str_A3   DCD       0xFFFFFF00     ;putting 0xFF after the string
15          ;Start    of program
16 Start    MOV       R1,#Count_A    ;initialise pointer R1 to mem var Count_A
17          ADD       R2,R1,#4       ;initialise pointer R2 to start of String_A
18          MOV       R0,#0          ;clear R0
19          STR       R0,[R1]        ;clear mem var Count_A before counting
20 Loop     LDRB      R3,[R2]        ;get current element in string from memory
21          CMP       R3,#Null_char  ;compare with Null value
22          BEQ       Done           ;branch to done if Null character
23          LDR       R4,[R1]        ;get mem var Count_A
24          ADD       R4,R4,#1       ;increment string length count
25          STR       R4,[R1]        ;save new length back to mem var Count_A
26          ADD       R2,R2,#1       ;increment pointer to next string element
27          B         Loop           ;loop back to loop
28 Done     END
```

Figure 7 – Listing of the pre-optimised *Expt_1d_Application_2* ARM assembly program.

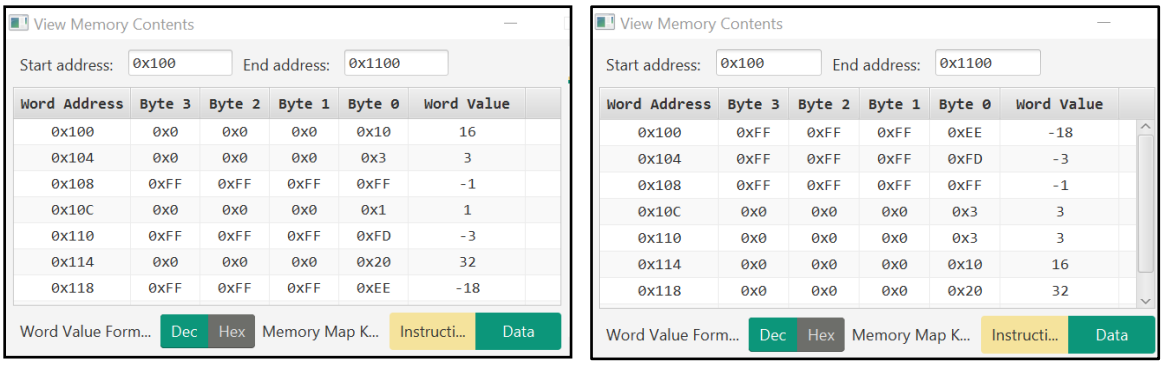
Figure 7b – My optimized version and the total cycle count is

5.6 Write your own ARM assembly program

Figure 8 shows the template of the ARM assembly program with an unordered array of seven 32-bit integers stored starting at the label **`N_Array`**. Write a program that will sort this array into ascending order with the smallest integer value at the lowest address in the array (i.e. at address label **`N_array`**). Do note that integers are signed values and negative values are smaller than positive values. Your initial memory map in Figure 9(a) should turn into Figure 9(b) after execution.

```
1           ;-------------------------------------------------------------
2           ;Expt_1e  Application #3 - Sorting into ascending order
3           ;1.       Sort the 7 integers in array N-Array in ascending order
4           ;2.       The smallest value starts are N_Array after execution
5           ;-------------------------------------------------------------
6           ;Allocate variables and constants into data memory
7 DataArea  EQU       0x00000100     ;data area starts at 0x0100
8 N_Size    EQU       7              ;declare size of N-Array
9 N_Array   DCD       0x00000010     ;start of N-array, decimal (16)
10 N_2      DCD       0x00000003     ;decimal (3)
11 N_3      DCD       0xFFFFFFFF     ;decimal (-1)
12 N_4      DCD       0x00000003     ;decimal (3)
13 N_5      DCD       0xFFFFFFFD     ;decimal (-3)
14 N_6      DCD       0x00000020     ;decimal (32)
15 N_7      DCD       0xFFFFFFEE     ;decimal (-18)
16          ;Start    of program
17          END
```

Figure 8 – Listing of the incomplete *Expt_1e_Application_3* ARM assembly program.



(a) (b)

Figure 9 – (a) Memory map at the start of program execution and (b) memory after program execution has completed, where the 7 integers have been sorted in ascending order.

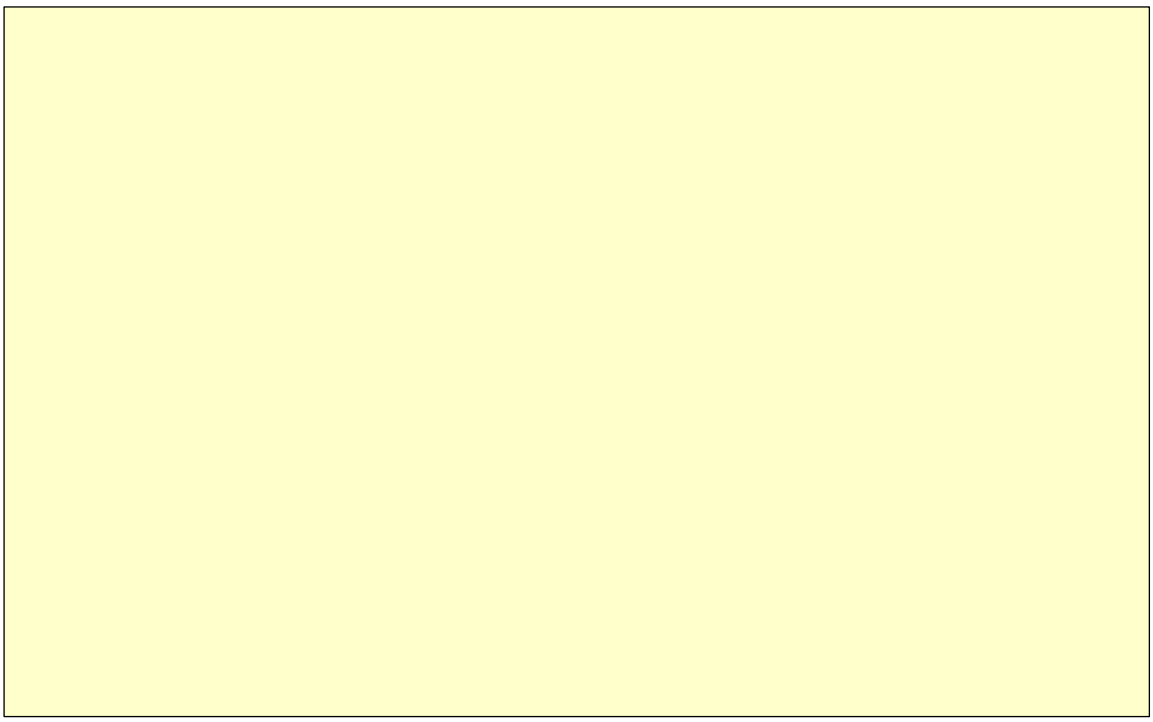


Figure 8b – My ARM assembly program for sorting an array in ascending order.

6. LOGGING RESULTS AND OBSERVATIONS

You can record your results, observations and analysis in section 5 into a hardcopy of your **Lab Manual** first or type directly into the pdf form version of the ***Lab_Manual_Expt_1*** (available on NTULearn Laboratory folder) during the actual lab session itself.

7. REFERENCES

7.1 CE1106/CZ1106 Lecture Notes on Addressing Modes by A/P Goh Wooi Boon (2020)
7.2 The VisUAL User Guide
7.3 The ARM instruction set subset supported by VisUAL