Assignment 3B (50 marks) – Lab Week Eleven – Fun with Arrays in Assembly Language

Submission Link for Lab Week 11: Assignment 3B

Due Date: By 11:30 Friday, 30 July 2021 using the Submission Link for Lab Week 11: Assignment 3B

Early Submissions Welcome!

Late submissions will not be accepted and will receive a mark of zero (0). If your name is not on the Hand-In Sheet or the Code Listings or the code is not adequately documented, you will not receive credit for this assignment.

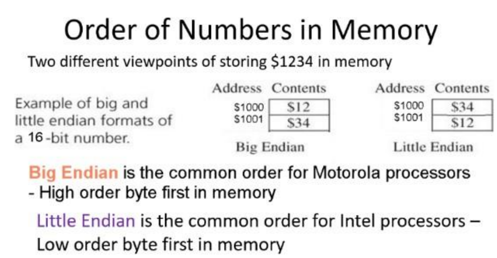
This lab exercise may be optionally performed by up to **THREE** students **in the same lab section** working as a group (students pick their own partners). Also, ensure all student names/numbers are on all program listings and documentation in order for all students to receive credit for the work (No name, no credit). **There is only ONE submission required per group of students, so ensure that your student group coordinates who is responsible for submitting your code and add student in Brightspace submission link message.**

PURPOSE OF LAB:

*The purpose of this lab is to gain more experience with Arrays in Assembly Language by using **AsmIDE** and the **Dragon12 & Student Mode Simulator** to create and test software that will manipulate the content of data in memory.*
The tasks in this lab are a follow-on to Week 10's Lecture where we discussed various Addressing Modes and how copy to copy an Array using problem solving methods for Assembly Language. The use of Memory Maps and a Flowchart to illustrate the problem domain were emphasized.

This is a multipart Assignment, where each Task will build upon the previous one; however, students who do not successfully implement each Task may receive partial credit for their work.

Task One (10 Marks) – The Little Endian Challenge (Endian.asm)



In Week Four of the course, we explored the concept of "The Endians." The purpose of this task is to confirm your understanding on how to write assembly language code to copy data from one memory location to another while at the same time swapping the values so that the data is changed from Intel 16-bit Little Endian to Motorola 16-bit Big Endian formats. This Task will also assist you in better understand Iteration, the capabilities of the auto-post increment Indexed (IDX) Addressing mode and HCS12 Transfer instructions. Most of the theory behind this task was explored in Week Ten's Lecture.

Procedure:

- Using the supplied skeleton code (partial code listing) **Endian.asm,** write a complete HCS12 Assembly program that effectively uses **Iteration** and **Pointers** to convert the given 16-bit **UNSIGNED** Little Endian data to 16-bit Big Endian data as per the following Pre-Execution and Post-Execution Memory Maps. Note that this means you must copy 16 bits of data at a time from one memory location to another. Use **Big_Endian** and **End_Big_Endian** as labels for the start/end of the new array. This will facilitate dynamic array length calculations.
- Don't forget to include Your Name(s) and Student Numbers(s) and Modification Date in the program header.
- The HCS12 Instruction set that you will likely use in this Task in the IMM, INH and IDX Addressing modes are: **ldx, ldy, ldd, std, cpx** and **an UNSIGNED branch**. As such, you should review their use as taught in class and in the Almy Text and the S12CPUV2 Reference Manual.
- Include documentation of your code that explains WHAT your code does, NOT the instruction set
- Note that the optimal solution should contain no more than **7** lines of code between initiating the Stack and the **swi** statement
- In order for your solution to be 100% functional, your solution must realize the following Pre-Execution and Post-Execution Memory Maps, which illustrate the original Little Endian data (starts at $1000) followed by the copied/converted Big Endian format of data (starts at $1010).

Pre-Execution Memory Map

Memory Display Address 1000 Show

ADDR	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1000	12	34	28	88	aa	55	00	ff	ff	00	55	aa	01	01	ff	00
1010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
1020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Post-Execution Memory Map

Memory Display Address 1000 Show

ADDR	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1000	12	34	28	88	aa	55	00	ff	ff	00	55	aa	01	01	ff	00
1010	34	12	88	28	55	aa	ff	00	00	ff	aa	55	01	01	00	ff
1020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Task Two (15 Marks) – Reversing Eight Bit Data Elements in an Array

In Week Ten of the course, we explored the concept of the auto-post increment Indexed (IDX) Addressing mode. The purpose of this task is to confirm your understanding on how to write assembly language code to copy data from one memory location to another while at the same time reversing the array elements. It will also assist you in better understanding Iteration, the capabilities of the various auto-pre/post increment Indexed (IDX) Addressing modes. Now that we have transformed the 16-bit Little Endian data into 16-bit Big Endian data, will now treat the data like normal 8-bit values for this Task and other Tasks in this assignment.

Most of the theory behind this task was explored in Week Ten's Lecture, amplified by Week Nine's Hybrid Lecture on Arrays as well the Almy Text and the S12CPUV2 Reference Manual.

Procedure:

- **OPTION A**: If your program in Task One is 100% functional, then save **Endian.asm** and then save it again as **BIG_E_Reverse.asm.** Build upon Task One's Code in **BIG_E_Reverse.asm** – e.g. add more lines of code after the code used to convert the Little → Big Endian data.
- Place the following comment just before your new code: ; --- Start of Copy and Reverse Big Endian Array Code ---
- **OPTION B:** If your program in Task One is not 100% functional, then save **Endian.asm** and submit it for partial marks. Then create a new program called **Reverse.asm** and use the following 8-bit data array values coded to originate at $1010 in your solution: $34, $12, $88, $28, $55, $AA, $FF, $00, $00, $FF, $AA, $55, $01, $01, $00, $FF.
- **FOR BOTH OPTIONS**: Your new program must effectively use iteration and pointers to copy one byte of data at a time from the Big Endian array to another memory location that starts immediately after the label **End_Little_Endian**, reversing the order of the Big Endian Array' data you earlier created. Use **Reverse** and **End_Reverse** as labels for the start/end of the new array. This will facilitate dynamic array size calculations. The Pre-Execution and Post-Execution Memory Maps are included below.
- **Do NOT hard code any Array Lengths – use Dynamically calculated array lengths in your solution as taught in class.**
- Recall that you can "Add" values to labels after an instruction, as taught in class. For example, your solution may include code such as: **ldy #End_Big_Endian-1** or simply **ldy #End_Big_Endian** depending on the program logic you use.
- Don't forget to include Your Name(s) and Student Numbers(s) and Modification Date in the program header
- The HCS12 Instruction set that you will likely use in this Task in the IMM, INH and IDX Addressing modes are: **ldx, ldy, ldaa, staa, cpy or cpx** and **an UNSIGNED branch**. As such, you should review their use as taught in class and in the Almy Text and the S12CPUV2 Reference Manual.
- Include documentation of your code that explains WHAT your code does, NOT the instruction set
- Note that the optimal solution should contain no more than 6 additional lines of code after your last line of Task One's program and the swi statement
- In order for your solution to be 100% functional, your solution must realize the following Pre-Execution and Post-Execution Memory Maps, which illustrates the original Little Endian data (starts at $1000) followed by the copied/converted Big Endian format of data (starts at $1010), followed by the reversed data (starts at $1020).

Pre-Execution Memory Map – Option A and B

Memory Display Address 1000 Show

ADDR	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1000	12	34	28	88	aa	55	00	ff	ff	00	55	aa	01	01	ff	00
1010	34	12	88	28	55	aa	ff	00	00	ff	aa	55	01	01	00	ff
1020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Post-Execution Memory Map – Option A and B

Memory Display Address 1000 Show

ADDR	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
1000	12	34	28	88	aa	55	00	ff	ff	00	55	aa	01	01	ff	00	|
1010	34	12	88	28	55	aa	ff	00	00	ff	aa	55	01	01	00	ff	|
1020	ff	00	01	01	55	aa	ff	00	00	ff	aa	55	28	88	12	34	|
1030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	|

Task Three (25 Marks) – Sorting Eight Bit Data Elements in an Array

The purpose of this task is to confirm your understanding on how to write assembly language code to search through an array and reorganize the elements so that they are in a particular order. It will also assist you in better understanding Iteration, the capabilities of the various auto-pre/post increment Indexed (IDX) Addressing modes, Unsigned branches, various Inherent instructions and the use of Labels with meaningful names that enhance the professionalism of your coding solution. Now that we have reversed the order of our 8-bit data in the previous Task, it is time to sort this **Unsigned** data from its lowest to its highest value, keeping in mind that **Signed branches are NEVER used with Unsigned data, even if the code "appears" to work!**
Most of the theory behind this task will be explored in Week Twelve's Lecture, however, sorting an array should not be a new concept to Level Three students in your program of study.

Finally, to assist you with this task, there is a Insertion Sort Video, an A3B All Tasks Video, and an Insertion_sort_from_Wikipedia document on Brightspace. I also found an interesting link to that may benefit your understanding of sorting: "Link to Problem Solving: The Insertion Sort"
https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheInsertionSort.html

Procedure:

- **OPTION A**: If your program in Task Two is 100% functional, then save **BIG_E_Reverse.asm** and then save it again as **BIG_E_Reverse_Sorted.asm.** Build upon Task Two's Code in **BIG_E_Reverse_Sorted.asm** – e.g. add more lines of code after the code used to convert the copy the array and place its reversed version in a new memory location.
- Place the following comment just before your new code: ; --- Start of Insertion Sort Code ----
- **OPTION B:** If your program in Task Two is not 100% functional, then save **Reverse.asm** and submit it for partial marks. Then create a new program called **Sorted.asm** and use the following 8-bit data array values coded to originate at $1010 in your solution: $34, $12, $88, $28, $55, $AA, $FF, $00, $00, $FF, $AA, $55, $01, $01, $00, $FF **and** use the following 8-bit data array values coded to originate at $1020 in your solution: $FF, $00, $01, $01, $55, $AA, $FF, $00, $00, $FF, $AA, $55, $28, $88, $12, $34
- **FOR BOTH OPTIONS**: Your new program must effectively use iteration and pointers to parse the array and sort the **unsigned** 8-bit data from its lowest to highest value. Note that you **must sort the array "in-place." Do NOT make another copy of the array, sort it, then overwrite the original array with its sorted values. If you do, you can expect a significant mark reduction.**
- **Do NOT hard code any Array Lengths – use Dynamically calculated array lengths in your solution as taught in class.**
- Recall that you can "Add" values to labels after an instruction, as taught in class. For example, your solution may include code such as: **ldy #End_Big_Endian-1** or simply **ldy #End_Big_Endian** depending on the program logic you use.
- Don't forget to include Your Name(s) and Student Numbers(s) and Modification Date in the program header
- The HCS12 Instruction set that you will likely use in this Task in the IMM, INH and IDX Addressing modes are: **ldx, ldy, tfr, dey or dex, inx or iny, ldaa, staa, cpy or cpx** and **an UNSIGNED branch**. As such, you should review their use as taught in class and in the Almy Text and the S12CPUV2 Reference Manual.
- Include documentation of your code that explains WHAT your code does, NOT the instruction set
- Note that the optimal solution should contain about **18-20** additional lines of code after your last line of Task Two's program and the swi statement
- In order for your solution to be 100% functional, your solution must realize the following Pre-Execution Memory Map, which illustrates the original Little Endian data (starts at $1000) followed by the copied/converted Big Endian format of data (starts at $1010), followed by the reversed data (starts at $1020). Then, after the Insertion Sort, realizes the Post-Execution Memory Map which illustrates the original Little and Big Endian data **and** the sorted array (starts at $1020).

Pre-Execution Memory Map – Option A and B

Memory Display Address 1000 Show

ADDR	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
1000	12	34	28	88	aa	55	00	ff	ff	00	55	aa	01	01	ff	00	|
1010	34	12	88	28	55	aa	ff	00	00	ff	aa	55	01	01	00	ff	|
1020	ff	00	01	01	55	aa	ff	00	00	ff	aa	55	28	88	12	34	|
1030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	|

Post-Execution Memory Map – Option A and B

Memory Display Address 1000 Show

ADDR	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
1000	12	34	28	88	aa	55	00	ff	ff	00	55	aa	01	01	ff	00	|
1010	34	12	88	28	55	aa	ff	00	00	ff	aa	55	01	01	00	ff	|
1020	00	00	00	01	01	12	28	34	55	55	88	aa	aa	ff	ff	ff	|
1030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	|

Assignment 3B (50 marks) – Lab Week Eleven – Fun with Arrays in Assembly Language

Submission Link for Lab Week 11: Assignment 3B

Assessment details.

Assessment

It is highly recommended that you check your solutions against the following and marking rubric. BEFORE submitting your code and the Hand-In Sheet. Note that your submission will be assessed using the following rubric

Task One (10 Marks): Endian.asm

a. Student information present in program header
b. copied 16 bits (a word) of data at a time
c. effectively used Iteration and Pointers
d. effectively used Labels, NOT Absolute Memory Addresses
e. used Big_Endian and End_Big_Endian labels to dynamically calculate array size
f. effectively used branch statements and did NOT use SIGNED branch statements
g. code documentation explains WHAT code does, NOT the instruction set
h. Pre/Post Execution Memory Maps identical to provided ones (100% functional)

Note: If Student Information is missing or Iteration/Pointers were not used or code documentation is missing, no credit will be given for this Task.

Task Two (15 Marks): BIG_E_Reverse.asm OR Reverse.asm

a. array used to reverse data from Task One starts immediately after Lab End_Big_Endian
b. effectively used Iteration and Pointers
c. effectively used Labels, NOT Absolute Memory Addresses
d. used Reverse and End_Reverse labels to dynamically calculate the array size
e. effectively used branch statements and did NOT use SIGNED branch statements
f. code documentation explains WHAT code does, NOT the instruction set
g. Pre/Post Execution Memory Maps identical to provided ones (100% functional)

Note: If Student Information is missing or Iteration/Pointers were not used or code documentation is missing, no credit will be given for this Task.

Task Three (25 Marks): BIG_E_Reverse_Sorted.asm OR Sorted.asm

a. sorted array occupies same memory as unsorted array from Task Two
b. there was no use of an additional array in the sorting process
c. effectively used Iteration and Pointers with Reverse and End_Reverse labels
d. used meaningful Label names such as Read, DoWhile, Move_To_Right, Store, Done; NOT Label Names such as: A, R, S and so on
e. used Reverse and End_Reverse labels to dynamically calculate the array size
f. effectively used branch statements and did NOT use SIGNED branch statements
g. code documentation explains WHAT code does, NOT the instruction set
h. Pre/Post Execution Memory Maps identical to provided ones (100% functional)

Note: If Student Information is missing or Iteration/Pointers were not used or code documentation is missing, no credit will be given for this Task.

Assignment 3B (50 marks) – Lab Week Eleven – Fun with Arrays in Assembly Language

Criteria	Level 6	Level 5	Level 4	Level 3	Level 2	New Level 1	Criterion Score
Criterion 1 Task One (10 Marks): Endian.asm	10 points a. Student information present in program header b. copied 16 bits (a word) of data at a time c. effectively used Iteration and Pointers d. effectively used Labels, NOT Absolute Memory Addresses e. used Big_Endian and End_Big_Endian labels to dynamically calculate array size f. effectively used branch statements and did NOT use SIGNED branch statements g. code documentation explains WHAT code does, NOT the instruction set h. Pre/Post Execution Memory Maps identical to provided ones (100% functional)	8 points One of following is not done effectively : b. copied 16 bits (a word) of data at a time c. effectively used Iteration and Pointers d. effectively used Labels, NOT Absolute Memory Addresses e. used Big_Endian and End_Big_Endian labels to dynamically calculate array size f. effectively used branch statements and did NOT use SIGNED branch statements g. code documentation explains WHAT code does, NOT the instruction set	6 points Two of following are is not done effectively : b. copied 16 bits (a word) of data at a time c. effectively used Iteration and Pointers d. effectively used Labels, NOT Absolute Memory Addresses e. used Big_Endian and End_Big_Endian labels to dynamically calculate array size f. effectively used branch statements and did NOT use SIGNED branch statements g. code documentation explains WHAT code does, NOT the instruction set	4 points Three of following are not done effectively : b. copied 16 bits (a word) of data at a time c. effectively used Iteration and Pointers d. effectively used Labels, NOT Absolute Memory Addresses e. used Big_Endian and End_Big_Endian labels to dynamically calculate array size f. effectively used branch statements and did NOT use SIGNED branch statements g. code documentation explains WHAT code does, NOT the instruction set	2 points Four of following are not done effectively : b. copied 16 bits (a word) of data at a time c. effectively used Iteration and Pointers d. effectively used Labels, NOT Absolute Memory Addresses e. used Big_Endian and End_Big_Endian labels to dynamically calculate array size f. effectively used branch statements and did NOT use SIGNED branch statements g. code documentation explains WHAT code does, NOT the instruction set	0 points If following are missing: a. Student information present in program header b. Used Iteration and Pointers f. code documentation explains WHAT code does, NOT the instruction set g. Pre/Post Execution Memory Maps identical to provided ones (100% functional) h. Your submission	/ 10
Criterion 1 Task Two (15 Marks): BIG_E_Reverse.asm OR Reverse.asm	15 points a. array used to reverse data from Task One starts immediately after Lab End_Big_Endian b. effectively used Iteration and Pointers c. effectively used Labels, NOT Absolute Memory Addresses d. used Reverse and End_Reverse labels to dynamically calculate the array size e. effectively used branch statements and did NOT use SIGNED branch statements f. code documentation explains WHAT code does, NOT the instruction set g. Pre/Post Execution Memory Maps identical to provided ones (100% functional)	12 points One of following is not done effectively : a. array used to reverse data from Task One starts immediately after Lab End_Big_Endian b. effectively used Iteration and Pointers c. effectively used Labels, NOT Absolute Memory Addresses d. used Reverse and End_Reverse labels to dynamically calculate the array size e. effectively used branch statements and did NOT use SIGNED branch statements f. code documentation explains WHAT code does, NOT the instruction set	10 points Two of following are not done effectively : a. array used to reverse data from Task One starts immediately after Lab End_Big_Endian b. effectively used Iteration and Pointers c. effectively used Labels, NOT Absolute Memory Addresses d. used Reverse and End_Reverse labels to dynamically calculate the array size e. effectively used branch statements and did NOT use SIGNED branch statements f. code documentation explains WHAT code does, NOT the instruction set	8 points Three of following are not done effectively : a. array used to reverse data from Task One starts immediately after Lab End_Big_Endian b. effectively used Iteration and Pointers c. effectively used Labels, NOT Absolute Memory Addresses d. used Reverse and End_Reverse labels to dynamically calculate the array size e. effectively used branch statements and did NOT use SIGNED branch statements f. code documentation explains WHAT code does, NOT the instruction set	4 points Four of following are not done effectively : a. array used to reverse data from Task One starts immediately after Lab End_Big_Endian b. effectively used Iteration and Pointers c. effectively used Labels, NOT Absolute Memory Addresses d. used Reverse and End_Reverse labels to dynamically calculate the array size e. effectively used branch statements and did NOT use SIGNED branch statements f. code documentation explains WHAT code does, NOT the instruction set	0 points If following are missing: a. Student information present in program header b. Used Iteration and Pointers f. code documentation explains WHAT code does, NOT the instruction set g. Pre/Post Execution Memory Maps identical to provided ones (100% functional) h. Your submission	/ 15
Task Three (25 Marks): BIG_E_Reverse_Sorted.asm OR Sorted.asm	25 points a. sorted array occupies same memory as unsorted array from Task Two b. there was no use of an additional array in the sorting process c. effectively used Iteration and Pointers with Reverse and End_Reverse labels d. used meaningful Label names such as Read, DoWhile, Move_To_Right, Store, Done; NOT Label Names such as: A, R, S ... e. used Reverse and End_Reverse labels to dynamically calculate the array size f. effectively used branch statements and did NOT use SIGNED branch statements g. code documentation explains WHAT code does, NOT the instruction set h. Pre/Post Execution Memory Maps identical to provided ones (100% functional)	20 points One of following is not done effectively : a. sorted array occupies same memory as unsorted array from Task Two b. there was no use of an additional array in the sorting process c. effectively used Iteration and Pointers with Reverse and End_Reverse labels d. used meaningful Label names such as Read, DoWhile, Move_To_Right, Store, Done; NOT Label Names such as: A, R, S ... e. used Reverse and End_Reverse labels to dynamically calculate the array size f. effectively used branch statements and did NOT use SIGNED branch statements g. code documentation explains WHAT code does, NOT the instruction set	15 points Two of following are not done effectively : a. sorted array occupies same memory as unsorted array from Task Two b. there was no use of an additional array in the sorting process c. effectively used Iteration and Pointers with Reverse and End_Reverse labels d. used meaningful Label names such as Read, DoWhile, Move_To_Right, Store, Done; NOT Label Names such as: A, R, S ... e. used Reverse and End_Reverse labels to dynamically calculate the array size f. effectively used branch statements and did NOT use SIGNED branch statements g. code documentation explains WHAT code does, NOT the instruction set	10 points Three of following are not done effectively : a. sorted array occupies same memory as unsorted array from Task Two b. there was no use of an additional array in the sorting process c. effectively used Iteration and Pointers with Reverse and End_Reverse labels d. used meaningful Label names such as Read, DoWhile, Move_To_Right, Store, Done; NOT Label Names such as: A, R, S ... e. used Reverse and End_Reverse labels to dynamically calculate the array size f. effectively used branch statements and did NOT use SIGNED branch statements g. code documentation explains WHAT code does, NOT the instruction set	5 points Four of following are not done effectively : a. sorted array occupies same memory as unsorted array from Task Two b. there was no use of an additional array in the sorting process c. effectively used Iteration and Pointers with Reverse and End_Reverse labels d. used meaningful Label names such as Read, DoWhile, Move_To_Right, Store, Done; NOT Label Names such as: A, R, S ... e. used Reverse and End_Reverse labels to dynamically calculate the array size f. effectively used branch statements and did NOT use SIGNED branch statements g. code documentation explains WHAT code does, NOT the instruction set	0 points If following are missing: a. Student information present in program header b. Used Iteration and Pointers f. code documentation explains WHAT code does, NOT the instruction set g. Pre/Post Execution Memory Maps identical to provided ones (100% functional) h. Your submission	/ 25

Total	/ 50

