Assignment 4 (100 marks) – Lab Week Eleven (Due: See Hand-In Sheet).

This lab exercise may be optionally performed by THREE students working as a group (students pick their own partners from the SAME lab section). If you are in a multi-student group, 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).

Late Task One or Task Two submissions will not be accepted and will receive a mark of zero (0).

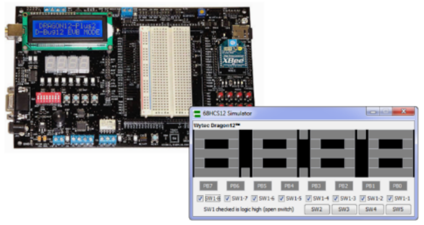
Additionally, you must be present to receive the marks for the in-lab demonstration of your solution.
Note: All students in the group receive the same mark for the assignment.

Download **Assignment4_zip**, which contains some skeleton code that will be used for this Task Three of this assignment.

PURPOSE OF LAB:
The purpose of this lab is to gain further experience in Assembly Language using ***AsmIDE*** with the ***Dragon12 & Student Mode*** *and the* ***Dragon 12 PLUS Trainer*** hardware board.

Additionally, we will deepen our knowledge of problem solving, hardware programing with the use of arrays, iteration, the Stack and HCS12 Addressing Modes.

Displaying Values on the Dragon12 Plus LEDs and the Simulator

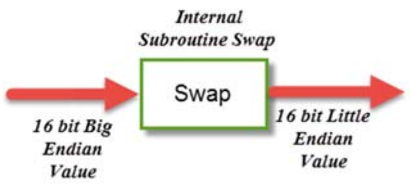


Task One – The Little Endian Challenge – Demo Solution in Simulator (15 marks)
Due: By the end of your week Thirteen's lab period (Week of 6 Apr 2015). (See note about being late).

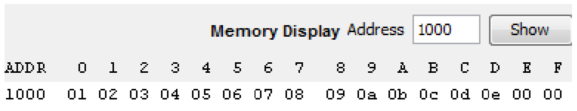
Building upon your knowledge of iteration in assembly language and supporting lecture material/your lecture notes and the program you wrote in A3A – Lab Week Nine, **A3_*Array.asm***, you are to write a complete assembly language program **(Swapper.asm)** that takes an array that is stored as Big Endian data and converts it to Little Endian data.

Notes:

a. Data starts at $1000, Code starts at $2000
b. Initialize the stack to $2000
c. Using iteration, take two values from the original array at a time. Within your program, write an internal subroutine called **Swap**. Using **only Stack operations within Swap**, convert the 16 bit Big Endian Value to a 16 bit Little Endian Value and then return to the main program and store back into the array, overwriting the original values.

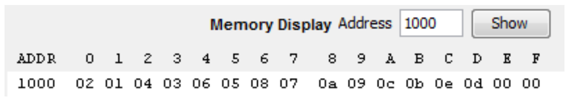


Pre-Execution Memory Map:



```
ADDR  0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
1000  01 02 03 04 05 06 07 08  09 0a 0b 0c 0d 0e 00 00
```

Post-Execution Memory Map:



```
ADDR  0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
1000  02 01 04 03 06 05 08 07  0a 09 0c 0b 0e 0d 00 00
```

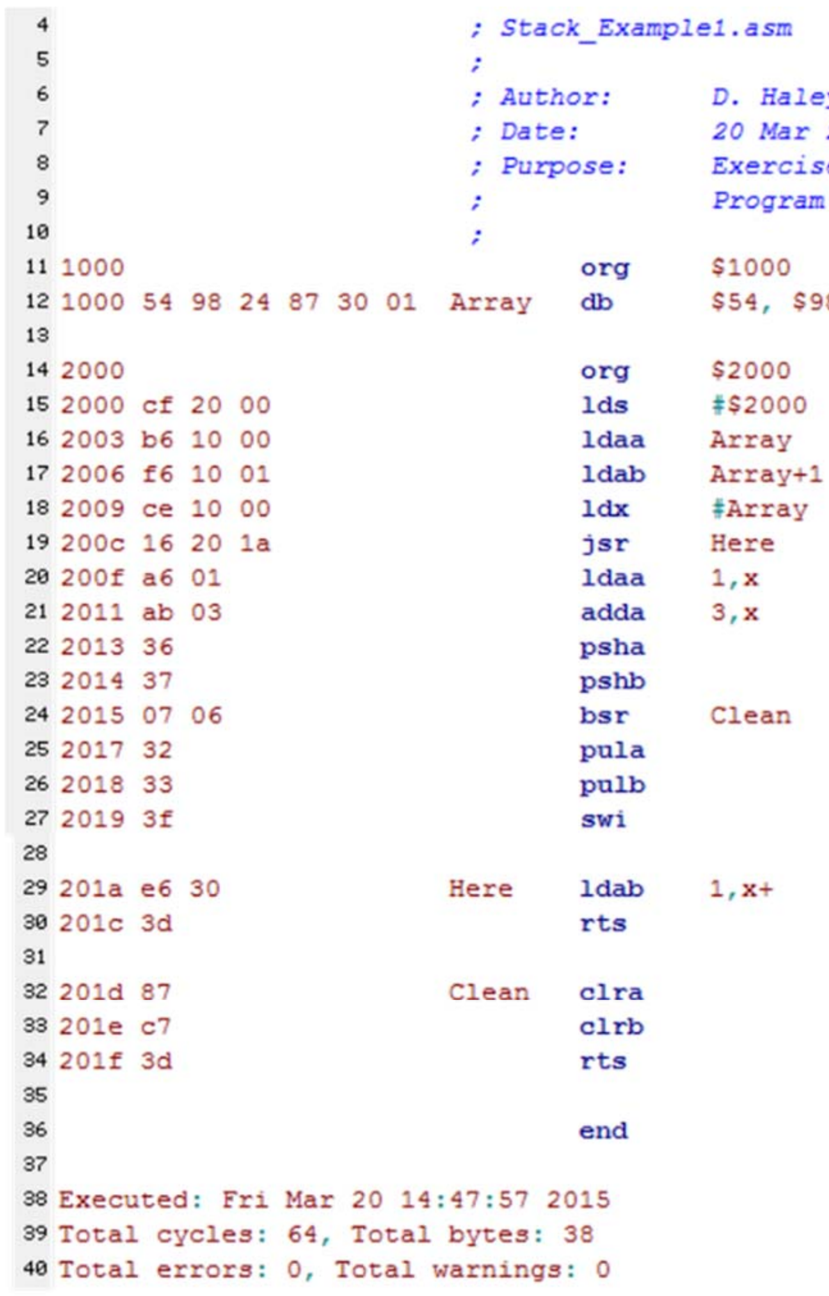
To complete this task (See Lab Hand-In Sheet):

a. Demo in simulator (5 marks);
b. In-class code inspection (10 marks)

Task Two – Program Tracing Involving the Stack (15 marks)

Due: By the end of your week Thirteen's lab period (Week of 6 Apr 2015). (See note about being late).

Given the following partial .lst file for ***Stack_Example1.asm***, complete the table for Task Two on the Hand-In Sheet by filling in the register/memory values that are present **after** the given Line of Code (LOC) is executed.



```
 4                                 ; Stack_Example1.asm
 5                                 ;
 6                                 ; Author:      D. Haley, Faculty
 7                                 ; Date:        20 Mar 2016
 8                                 ; Purpose:     Exercise students' knowledge of
 9                                 ;              Program flow and the Stack Pointer
10                                 ;
11 1000                                   org     $1000
12 1000 54 98 24 87 30 01  Array          db      $54, $98, $24, $87, $30, $01
13
14 2000                                   org     $2000
15 2000 cf 20 00                          lds     #$2000
16 2003 b6 10 00                          ldaa    Array
17 2006 f6 10 01                          ldab    Array+1
18 2009 ce 10 00                          ldx     #Array
19 200c 16 20 1a                          jsr     Here
20 200f a6 01                             ldaa    1,x
21 2011 ab 03                             adda    3,x
22 2013 36                                psha
23 2014 37                                pshb
24 2015 07 06                             bsr     Clean
25 2017 32                                pula
26 2018 33                                pulb
27 2019 3f                                swi
28
29 201a e6 30              Here           ldab    1,x+
30 201c 3d                                rts
31
32 201d 87                 Clean          clra
33 201e c7                                clrb
34 201f 3d                                rts
35
36                                        end
37
38 Executed: Fri Mar 20 14:47:57 2015
39 Total cycles: 64, Total bytes: 38
40 Total errors: 0, Total warnings: 0
```

Ensure all memory addresses in the table on the next page contain a value or " - - " if the value is not known at the time the line of code is executed.

DO NOT use $00 for unknown values.

Lab Week Eleven – Task Three Hand-in Sheet – Page Two
Hand this page in by the end of your week Thirteen's lab period (Week of 6 Apr 2015) – it will be emailed back to you completed.

Crazy Haley's Pizza Credit Card Validation Project – Demo Solution on the Wytec HCS12 Dragon12-Plus board (60 marks)

Your mark ________________________ /60

Name: ____ Student Number: ______________________

Name: ____ Student Number: ______________________

Name: ____ Student Number: ______________________

Due Dates:

a. Lab Demo – Software Demonstration on Wytec hardware board ideally by the end of your Lab Week 13 Lab Period (Week of 6 Apr 2015); however, you may also demo *without incurring a late penalty* during the two-hour lecture period (converted to a lab period) on Friday, 10 April 2015 (without late penalty).

b. Test Plan Submission – same timeframe as Lab Demo. Ensure copy goes into your portfolio folder.

c. Code Submission – A compressed file (see next page for details) containing a softcopy of all software listings and test plan uploaded to Blackboard's Assignment Four – Task Three submission link by 6:00 p.m. on Friday, 10 April 2015.

VIP

For Task Three - any late submissions receive a mark of zero (0) for that particular portion of the task. Therefore, if you do not get to demo by the demo deadline, you can still submit your code by the code deadline and be considered for full marks for that portion of the task. If you do not get the Test Plan completed by the end of the lab period, submit it with your code by its deadline so that it can be evaluated.

In-Lab Evaluation

In-Lab Demonstration (10 marks)

This is a "binary" mark – you must have a completed Test Plan to demonstrate and the results on the hardware board must match the Test Plan's expected Hex_Display subroutine output. If your software works in the simulator, but not on the hardware board, then it is likely because you did not initialize all of your registers and/or memory locations that you are using for temporary storage.

In-lab demonstration: /10 marks

Use the CC_Numbers.txt file that is applicable to your lab section. You must be present to receive an in-lab demonstration mark.

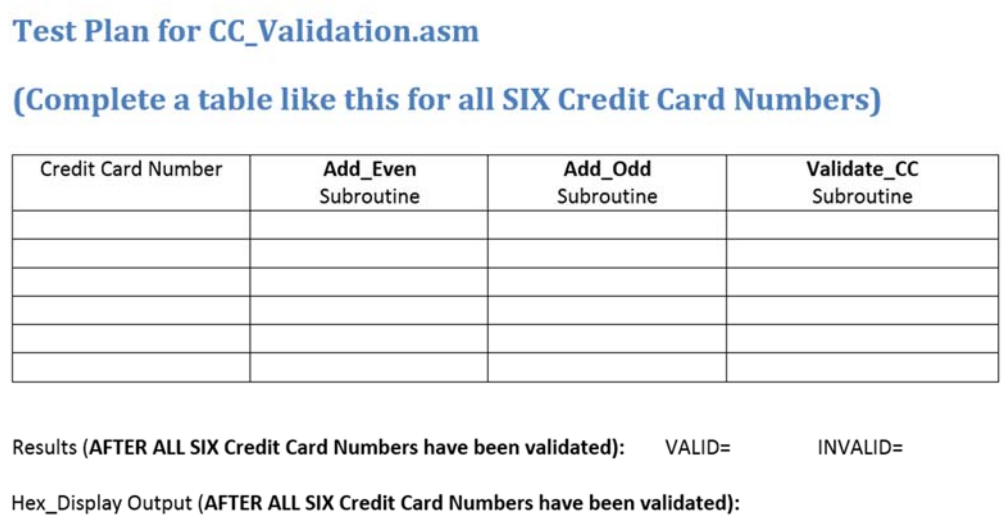
- Wednesday's Lab Group: 15W_WED_CC_Numbers.txt
- Thursday's Lab Group: 15W_THU_CC_Numbers.txt

In-Lab Submission

As well as using the hardcopy of your completed Test Plan to demonstrate the operation of your software, submit it to your portfolio before the end of your lab period so that I can also test your software post-lab. Also, include a copy with you on-line code submission.

Note 1: A partially completed Test Plan can be found within the compressed file supplied with this assignment.
Note 2: The Test Plan is to be created based upon ***your*** Credit Card Numbers file

- Wednesday's Lab Group: 15W_WED_CC_Numbers.txt
- Thursday's Lab Group: 15W_THU_CC_Numbers.txt



Test Plan for CC_Validation.asm

(Complete a table like this for all SIX Credit Card Numbers)

Credit Card Number	**Add_Even** Subroutine	**Add_Odd** Subroutine	**Validate_CC** Subroutine

Results (**AFTER ALL SIX Credit Card Numbers have been validated):** VALID= INVALID=

Hex_Display Output (**AFTER ALL SIX Credit Card Numbers have been validated):**

Lab Week Eleven – Task Three Hand-in Sheet – Page Three
Hand this page in by the end of your week Thirteen's lab period (Week of 6 Apr 2015) – it will be emailed back to you completed.

Post-Lab Assessment: Test Plan

a. The Test Plan will be assessed for completeness and accuracy /6 marks

Post-Lab Assessment: Code

a. **CC_Validation.asm** (including off-line assembly and test run on hardware board) /20 marks

b. **ADD_Even**.asm /12 marks

c. **Add_Odd**.asm /4 marks

d. **Validate**.asm /8 marks

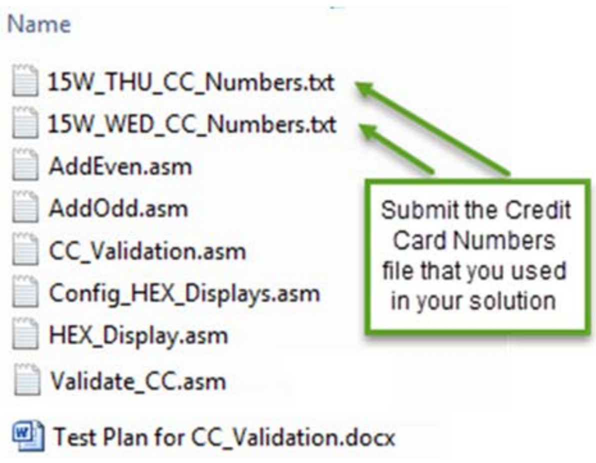
Note:

All code will be assessed as follows in descending priority:

a. Functionality – does the code correctly solve the problem? Is your code "spaghetti code?" Did you initialize all memory locations that will be used in the solution? (
b. Code conformance with the assignment requirements.
c. Assemble – does your submission assemble without errors or warnings?
d. Completeness – have you adequately documented your code?
e. Maintainability – wise use of iteration, formatting of code, use of assembler directives, labels and constants versus hardcoding values other than 0, 1, -1
f. Optimal use of the instruction set.
g. Documentation: program/subprogram title, header; pre/post conditions

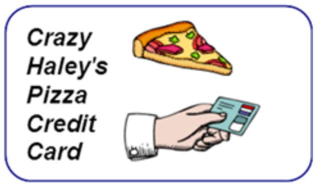
Required File Submission by 6:00 p.m. on Friday, 10 April 2015 (marks will be deducted for missing files)

- ✓ Place following files depicted to the right of this text into a single compressed .zip file upload to Blackboard's Assignment Four – Task Three submission link by 6:00 p.m. on Friday, 10 April 2015
- ✓ <u>Don't forget your Test Plan and Credit Card Numbers file</u> (Wednesday's Lab Group: 15W_WED_CC_Numbers.txt, Thursday's Lab Group: 15W_THU_CC_Numbers.txt)


Name
15W_THU_CC_Numbers.txt
15W_WED_CC_Numbers.txt
AddEven.asm
AddOdd.asm
CC_Validation.asm
Config_HEX_Displays.asm
HEX_Display.asm
Validate_CC.asm
Test Plan for CC_Validation.docx
Submit the Credit Card Numbers file that you used in your solution

Compressed File Naming Convention to use:
<Lab_Day>_<Student_Last_Names>.zip e.g. Wed_Bahair_Jones_McPhee.zip

All hand-in sheets will be emailed back to you with your mark and my comments.

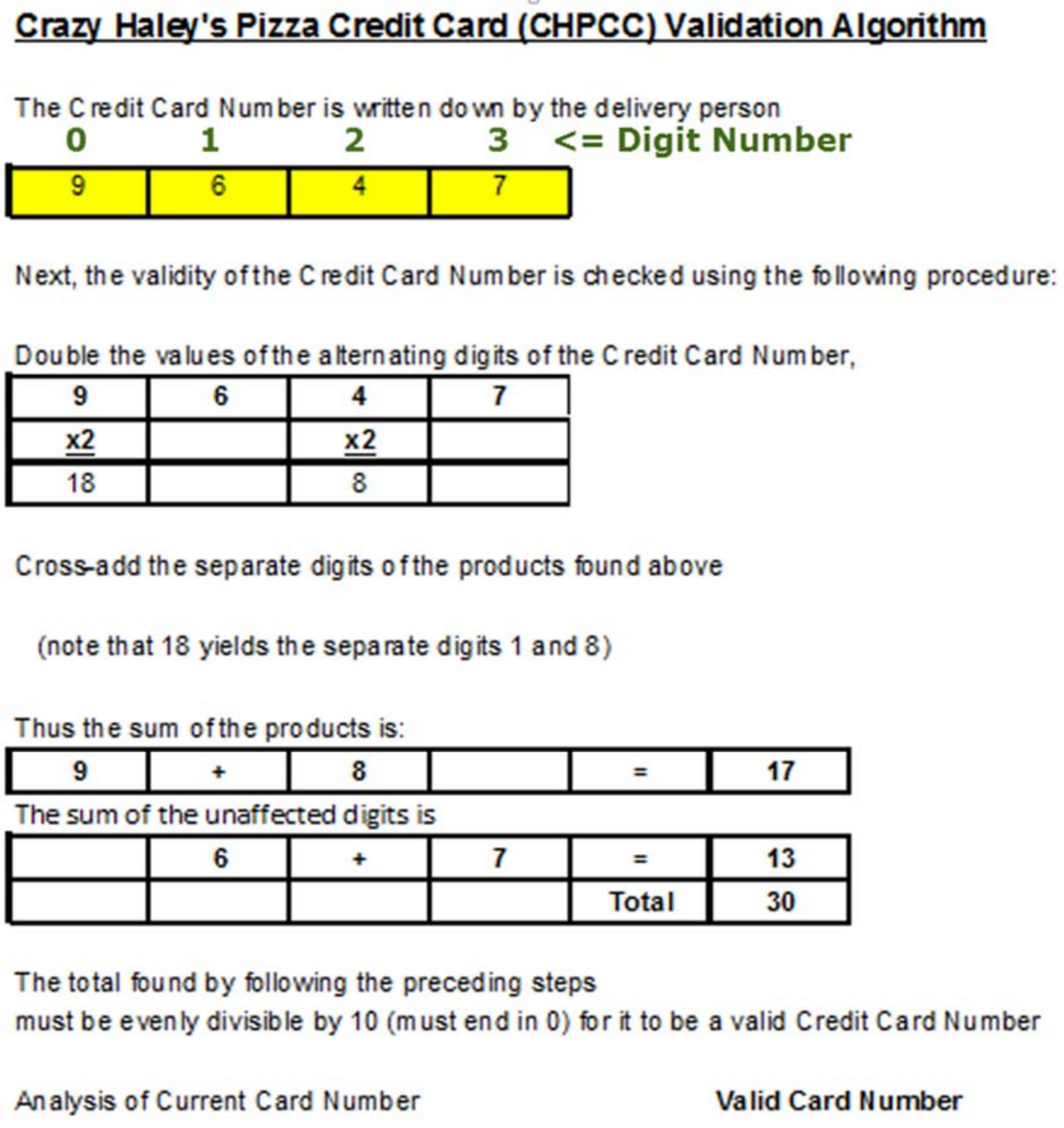


Congratulations! You have been contracted by Crazy Haley's Pizza to write a software application that automates the current manual process used to validate a Crazy Haley's Pizza Credit Card. These are unique credit cards used by *"thousands and thousands"* of students whose use their laptops in class to order pizza for everyone after their cell phones go off in class! The uniqueness of these credit cards is that the card numbers are 4 digits in length.

The Current Manual Process of Credit Card Validation

Once the delivery person has received a Crazy Haley's Pizza Credit Card from the student, a manual check of the card number is conducted using a pencil and paper to perform all of the calculations required to validate it.

An example of the calculations required follows, based upon the Luhn algorithm.



Crazy Haley's Pizza Credit Card (CHPCC) Validation Algorithm

The Credit Card Number is written down by the delivery person

0	1	2	3	<= Digit Number
9	6	4	7	

Next, the validity of the Credit Card Number is checked using the following procedure:

Double the values of the alternating digits of the Credit Card Number,

9	6	4	7
x2		x2	
18		8	

Cross-add the separate digits of the products found above

(note that 18 yields the separate digits 1 and 8)

Thus the sum of the products is:

9	+	8		=	17

The sum of the unaffected digits is

	6	+	7	=	13
				Total	30

The total found by following the preceding steps
must be evenly divisible by 10 (must end in 0) for it to be a valid Credit Card Number

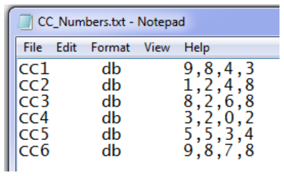
Analysis of Current Card Number **Valid Card Number**

The Future Automated Process of Credit Card Validation

MagTek MiniUSB Stripe Reader Card Reader

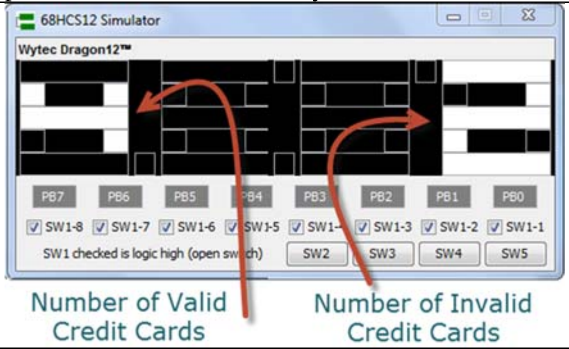
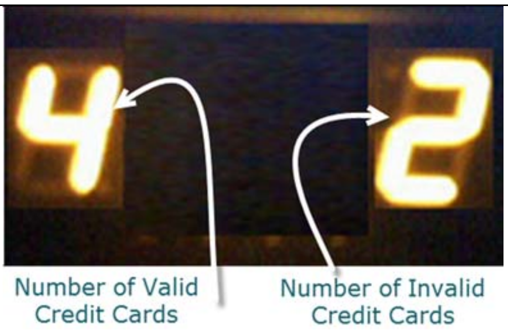
In the business of Software Engineering, many manual processes are automated to improve efficiency, risk management and customer satisfaction.

In a fully implemented system, we would swipe our credit cards using a device such as the one depicted to the right. However, before adding such a device to our system, we would first have to prove that the functionality of the software is correct. As such, we will simulate the actual reading of the credit card by the use of an external data file that contains exactly six credit card numbers, which we will read into our assembly language program.



```
CC1     db      9,8,4,3
CC2     db      1,2,4,8
CC3     db      8,2,6,8
CC4     db      3,2,0,2
CC5     db      5,5,3,4
CC6     db      9,8,7,8
```

We will validate each of six the credit card numbers, keeping track of the Number of Valid Credit Cards and the Number of Invalid Credit Cards as we iterate through the values . The following illustrates an example result.

Only ***after*** we determine the status of ***all*** of the six credit cards, will we will display the results on the 7-Segment Displays of the Simulator. Here is an **example** program run for the illustrated credit card number file, which will be provided to you to test your solution. You will also be provided with other files for your code demonstration.	Finally, once we have debugged our software and have confirmed that is FULLY functional, we will download it to the Dragon12-Plus HCS12 Trainer Board for deployment just like you did in previous labs.
 PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0 SW1-8 SW1-7 SW1-6 SW1-5 SW1-4 SW1-3 SW1-2 SW1-1 SW1 checked is logic high (open switch) Number of Valid Credit Cards Number of Invalid Credit Cards	

Development Requirements and Constraints

1. You must use the supplied skeleton code for ***CC_Validation.asm*** your starting point, and not change any of the code where annotated.

2. The first thing you should do is to create a test plan of your program run **BEFORE ANY CODE IS WRITTEN**. It must contain the calculations that prove the validity of your results using the CHPPC Validation Algorithm. Your test plan must be based upon the file of Credit Card numbers that you will "include" in your "main" program,
   - Wednesday's Lab Group: 15W_WED_CC_Numbers.txt
   - Thursday's Lab Group: 15W_THU_CC_Numbers.txt

   Note that have also supplied "CC_Numbers.txt", which contains the Credit Card numbers used in the example program run. I would recommend that you determine which numbers are valid/invalid in that file as well so that you can use that information to debug your program as you develop it. Then, once you understand what problem you are trying to solve and eventually code it, you know that the answer should be **4** and **2** as depicted above. Once you have that working, you can then use your own file of Credit Card numbers to validate your solution.

   Note that I will ask to see your test plan (and other rough work) should you run into difficulty with this assignment, as the test plan should reveal to you the inner workings of each of the subroutines.

3. Write a "main" Assembly Language program called ***CC_Validation.asm*** that determines the Number of Valid Credit Cards and the Number of Invalid Credit Cards as we iterate through the values Use the following high level program flow when writing ***CC_Validation.asm***, noting that there will be other things that "main" must also perform as it uses the named subprograms.

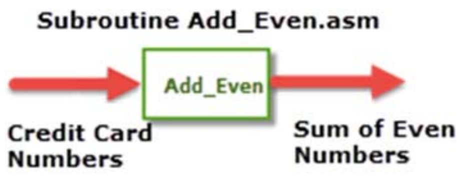
```
a. Initialize Program Constants and Variables;
b. Do
      i.   Call Add_Even
      ii.  Call Add_Odd
      iii. Call Validate_CC
   Until all six Credit Cards are validated

c. Call Config_HEX_Displays
d. Do
      i.   Call Hex_Display
      ii.  Call Delay_ms
   Forever
```

4. In support of ***CC_Validation.asm***, you must have "main" use three subroutines that manipulate the credit card digits and validate them based upon the example use of the Luhn algorithm illustrated in this document and the sample test plan provided to you. I have also provided you with a subroutine template.

5. As such, you must write the following Assembly Language subroutines and place them in your Source Code folder.

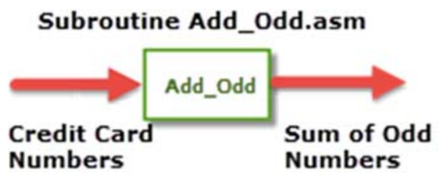
Add_Even

a. this subroutine is called by "main" using **jsr Add_Even**
b. Its purpose is to Add up the even numbered digits (digits 0 and 2), including the higher and lower nibbles if the number > 9
c. Its filename is ***Add_Even.asm***



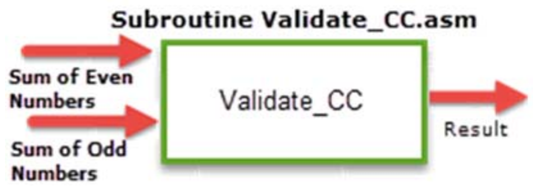
Add_Odd

a. this subroutine is called by "main" using **jsr Add_Odd**
b. Its purpose is to Add up the odd numbered digits (digits 1 and 3)
c. Its filename is ***Add_Odd.asm***



Validate

a. this subroutine is called by "main" using **jsr Validate_CC**
b. Its purpose is to add the Sum of the Even and Odd Numbers and determine if the sum is evenly divisible by 10. If it is, then the Credit Card number is valid; otherwise, it is not valid. **This** subroutine makes this decision, **not** "main"
c. Its filename is ***Validate_CC.asm***



Rules Regarding Subroutines created in separate files

i. Each subroutine is to be in a separate file and fully documented as per the documentation standards contained in ***Config_Hex_Displays.asm** and **Hex_Display.asm** which are supplied to you for this lab.*
ii. Subroutines are independent bodies of code and should have "no knowledge of each other." [e.g. In C, toupper() doesn't know about tolower()].
iii. Subroutines, including library files may only be called by "main."
iv. Subroutines must not contain an **org** statement.
v. While Subroutines have access to the global values contained in ***registers.inc***, subroutines cannot use "global" constants or other values defined by **equ** statements in "main." Rather, subroutines may contain their own **equ** statements for any values that are local to them.
vi. Subroutines are called by a **jsr** to the label that names the subroutine (look at ***Hex_Display.asm*** for an example).
vii. Subroutines must be "passed" pointers, and other values in registers from "main." These values must be documented in the Pre-Condition section of the subroutine header.
viii. Subroutines cannot modify memory in "main" or other subroutines' memory space. Rather, they must "return" values in registers. These values must be documented in the Post-Condition section of the subroutine header. Do not use the stack to return values to main.
ix. Within a subroutine, where any registers' contents are destroyed (overwritten with new values), this must be documented in the Post-Condition section of the subroutine header.

x. A subroutine may have one or more **rts** statements

HCS12 Instructions you are likely to use in this Lab Assignment

S12CPUV2 Reference Manual, Rev. 4.0

Table 5-1. Load and Store Instructions

Mnemonic	Function	Operation
	Load Instructions	
LDAA	Load A	(M) ⇒ A
LDAB	Load B	(M) ⇒ B
LDD	Load D	(M : M + 1) ⇒ (A:B)
LDS	Load SP	(M : M + 1) ⇒ SP_H:SP_L
LDX	Load index register X	(M : M + 1) ⇒ X_H:X_L
LDY	Load index register Y	(M : M + 1) ⇒ Y_H:Y_L

Table 5-2. Transfer and Exchange Instructions

Mnemonic	Function	Operation
	Transfer Instructions	
TAB	Transfer A to B	(A) ⇒ B

Table 5-6. Decrement and Increment Instructions

Mnemonic	Function	Operation
	Decrement Instructions	
DEC	Decrement memory	(M) – \$01 ⇒ M
DECA	Decrement A	(A) – \$01 ⇒ A
DECB	Decrement B	(B) – \$01 ⇒ B
	Increment Instructions	
INC	Increment memory	(M) + \$01 ⇒ M
INCA	Increment A	(A) + \$01 ⇒ A
INCB	Increment B	(B) + \$01 ⇒ B
INX	Increment X	(X) + \$0001 ⇒ X
INY	Increment Y	(Y) + \$0001 ⇒ Y

Table 5-9. Clear, Complement, and Negate Instructions

Mnemonic	Function	Operation
CLR	Clear memory	\$00 ⇒ M
CLRA	Clear A	\$00 ⇒ A
CLRB	Clear B	\$00 ⇒ B

Table 5-10. Multiplication and Division Instructions

Mnemonic	Function	Operation
	Division Instructions	
IDIV	16 by 16 integer divide (unsigned)	(D) ÷ (X) ⇒ X Remainder ⇒ D

Table 5-12. Shift and Rotate Instructions

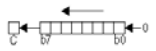
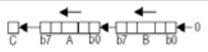
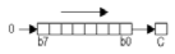
Mnemonic	Function	Operation
	Logical Shifts	
LSL LSLA LSLB	Logic shift left memory Logic shift left A Logic shift left B	
LSLD	Logic shift left D	
LSR LSRA LSRB	Logic shift right memory Logic shift right A Logic shift right B	

Table 5-21. Jump and Subroutine Instructions

Mnemonic	Function	Operation
BSR	Branch to subroutine	SP – 2 ⇒ SP RTN_H : RTN_L ⇒ $M_{(SP)}$: $M_{(SP+1)}$
JSR	Jump to subroutine	SP – 2 ⇒ SP RTN_H : RTN_L ⇒ $M_{(SP)}$: $M_{(SP+1)}$ Subroutine address ⇒ PC
RTS	Return from subroutine	$M_{(SP)}$: $M_{(SP+1)}$ ⇒ PC_H : PC_L SP + 2 ⇒ SP

Table 5-23. Index Manipulation Instructions

Mnemonic	Function	Operation
	Addition Instructions	
ABX	Add B to X	(B) + (X) ⇒ X
ABY	Add B to Y	(B) + (Y) ⇒ Y
	Compare Instructions	
CPX	Compare X to memory	(X) – (M : M + 1)
CPY	Compare Y to memory	(Y) – (M : M + 1)

Table 5-24. Stacking Instructions

Mnemonic	Function	Operation
	Stack Operation Instructions	
PSHA	Push A	(SP) – 1 ⇒ SP; (A) ⇒ $M_{(SP)}$
PSHB	Push B	(SP) – 1 ⇒ SP; (B) ⇒ $M_{(SP)}$
PSHX	Push X	(SP) – 2 ⇒ SP; (X) ⇒ $M_{(SP)}$: $M_{(SP+1)}$
PSHY	Push Y	(SP) – 2 ⇒ SP; (Y) ⇒ $M_{(SP)}$: $M_{(SP+1)}$
PULA	Pull A	($M_{(SP)}$) ⇒ A; (SP) + 1 ⇒ SP
PULB	Pull B	($M_{(SP)}$) ⇒ B; (SP) + 1 ⇒ SP

Your thought-process, coupled with flowcharts and your test plan should lead you to the various instructions that are typically used to code this assignment. For example, the given algorithm states that you must divide a certain result to see if the answer is evenly divisible by 10 – e.g. that the remainder is 0. Given that we are working with unsigned integer values, which should lead us to the ***idiv*** instruction. To see how this, or any instruction works, make up a small test program, set up the registers appropriately, execute the instruction and observe the results – e.g. make wise use of the simulator's debugging capabilities.

The same holds true for any code you develop – test it in the simulator – if you don't get the expected results, single step the code to discover where you went wrong.

As discussed in class, if you have worked out WHAT you want to do, but are stuck for an instruction to perform the task (given that you have already looked for an instruction), I would be happy to point you in the right direction. HOWEVER, I can only do that if you have a PLAN == detailed flowchart and a test plan and other rough work.