

**Western Michigan university**

Electrical and Computer Engineering department

**ECE-2510-560-Intro to microprocessor**

**LAB 3**

**S12(x) Assembler Directives and CCR**

Kaushik Mojumder and Sutopa Suhrid

Name of Laboratory Instructor- Husam Beitello  
Day/Time of your lab section-2.30 PM-5.10 PM, Thursday

**ECE-2100-540 - Circuit Analysis**

Date Laboratory was Performed-September 23,2021

Table of Contents

**PROCEDURES3**

Task 13

Task 28

Task 313

Task 421

**Analysis and Conclusion………………………………………………………………27**

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

1. **Procedures:**
   1. ***Task 1:***

In task 1, we basically used the variables to perform operations from a specific memory location. First we created a new project and renamed to lab3task1. Then we cleared the sample Fibonacci counter program and all the variables.

We compiled the code without any error and the no error dialogue also appeared. We added the following code under the variables declaration section:

ORG $1100;indicates the operation to start from m[1100]

num1: ds.b 1; indicates m[1100]

num2: ds.b 1;indicates m[1101]

Text

Description automatically generated with low confidence

Figure: task 1 variables

So we declared two variables num 1 and num 2 where num 1 allocates the byte starting from m[1100]. So num 2 allocated the byte at m[1101]. Then we added the following code under endless loop.

LDAA #$70 ; A <- $70

STAA num1 ; m[1100] <- $70

LDAA #$53 ; A<- $53

STAA num2 ; m[1101] <- $53

ADDA num1 ; [A] = $70 + $53 = $C3

STAA $1110; m[1110] <- $C3

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

The overall code looks like this:

Text

Description automatically generated with low confidence

Figure: source code for task 1

Then we compiled the code without any errors. The DRAGON12 Plus-2 was reset and the bootloader was uploaded. Then the program was debugged and we set a breakpoint to move step by step through each single line of code.

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

Graphical user interface, application, Word

Description automatically generated

Figure: LDAA #$70

Basically, the hexadecimal value $70 was stored in accumulator A

Graphical user interface, application, Word

Description automatically generated

Figure: STAA num1

Here the value of accumulator A was stored at memory location of num1. Previously we declared num 1 to allocate byte at m[1100]. So, now m[1100] = #$70.

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

Graphical user interface, text, application, Word

Description automatically generated

Figure: LDAA #$53

the hexadecimal value $53 was stored in accumulator A

Graphical user interface, application, Word

Description automatically generated

Figure: STAA num2

Num2 was declared to store the next byte at m[1101]. Here the value of accumulator A was stored at memory location of num2. So, now m[1101] = #$53

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

Graphical user interface, text, application

Description automatically generated

Figure: ADAA num1

The value stored at memory address pointed by num2 will be added to the value stored at memory address pointed by num1. The result will be stored at accumulator A. So, A = $70 + $53 = C3.

Graphical user interface, application, Word

Description automatically generated

Figure: STAA $1100

The value stored at A will also be stored at m[1100]. So, m[1100] = C3

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

Verification:

$70 + $53 = $C3

So, the value stored in the accumulator and memory addresses for this task are same.

* 1. ***Task 2:***

In this task, we declared constants instead of variables. First we declared a memory location from which the operations will start. Then we declared a constant which is more than 2 byte to store in the memory addresses from that location. Since, the value is more than 2 byte we need to use the write operation. The first memory will store the most significant bits and the next memory will store the least significant bits. First we created a new project and renamed to lab3task2. Then we cleared the sample Fibonacci counter program and all the variables.

We compiled the code without any error and the no error dialogue also appeared. We added the following code under the constant declaration section:

ORG $4100;indicates the operation to start from m[4100]

offset: dc.w 500; 500=$1F4.so,m[4100]=$01{msb} and m[4101]=$F4{lsb}

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

Text

Description automatically generated

Figure: task 2 constant section

Here, we declared a constant to store the decimal value 500. 500 = 01F4 in hexadecimal. Since it is greater than 2 byte so we used the write operation. As a result, more than 1 memory locations can be used to store the value. By ORG $4100, the operation already starts from m[4100]. So, m[4100] = $01(most significant bits) and m[4101] = $F4(least significant bits).

We added the following code under endless loop:

LDY #$1991;Y=$1991

LDX #$1000;X=$1000

LDD offset;D=$01F4

STY D,X;m[D+X]=11F4=Y

BRA EndlessLoop

The entire code looks like this:

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

Timeline

Description automatically generated with medium confidence

**.** figure: task 2 source code

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

Then we compiled the code without any errors. The DRAGON12 Plus-2 was reset and the bootloader was uploaded. Then the program was debugged and we set a breakpoint to move step by step through each single line of code.

Graphical user interface, text, application

Description automatically generated

Figure: LDY #$1991

The index register Y has an immediate value $1991. So, [Y] = $1991

Graphical user interface, application

Description automatically generated

Figure: LDX #$1000

The index register X has an immediate value $1000. So, [X] = $1000

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

Graphical user interface, application, Word

Description automatically generated

Figure: LDD offset

The accumulator D was loaded with a value of constant offset which is stored in m[4100] and m[4101]. So, [D] = $01F4

Graphical user interface, application, Word

Description automatically generated

Figure: STY D,X

D,X refers to a memory address equal to sum of load register D and index register X. It indicates a memory location of $01F4 + $1000= $11F4 and $01F4 + $1000 +$1=$11F5.

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

So, the value inside m[11F4] and m[11F5] will be stored in Y. The most significant bits will be from m[11F4] and least significant bits will be from m[11F5]. Since, m[11F4]= $19 and m[11F5] = $91, so Y= $1991.

Verification:

500 = $01F4

$01F4 + $1000 = $11F4

$01F4 + $1000 +$1 = $11F5

The hand calculated memory addresses contain the final value stored in index register Y.

* 1. ***Task 3:***

First we created a new project and renamed to lab3task3. Then we cleared the sample Fibonacci counter program and all the variables.

We compiled the code without any error and the no error dialogue also appeared. We added the following code under the constant declaration section:

ORG $1000;indicates the operation to start from m[1000]

offset: dc.b $A5,$A4,$20,$97; m[1000]=$A5, m[1001]=$A4,m[1002]=$20, m[1003]=$97

By using the ORG $1000, all the operations will start from m[1000].

We converted the decimal values to hexadecimal in the prelab as follows:

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

165 = $A5

164 = $A4

32 = $20

151 = $97

Since they are 2 bytes in size so, we used the byte operation to store these values starting from m[1000]. So, m[1000] = $A5, m[1001] = $A4, m[1002] = $32, m[1003] = $97.

Graphical user interface, text

Description automatically generated

Figure: task 3 constant section

Under the endless loop we added the following code:

LDX #$1000; X=m[1000]

LDAA 0,X;A=m[1000]+0=$A5

ADDA 1,X;A=m[1000]+m[1001]=$A5+$A4=$149

ADDA 2,X;A=A+m[1002]=$149+$20=$169

ADDA 3,X;A=A+m[1203]=$169+$97=$200

STAA $1210;m[1210]=$200

The overall code looks like this:

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

Text

Description automatically generated

Figure: task 3 source code

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

Then we compiled the code without any errors. The DRAGON12 Plus-2 was reset and the bootloader was uploaded. Then the program was debugged and we set a breakpoint to move step by step through each single line of code.

A screenshot of a computer

Description automatically generated

Figure: LDX #$1000

Here, index register is loaded with a hexadecimal value $1000. In binary

$1000 = 000100000000.

So, C, N, Z, V all will be disabled.

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

Graphical user interface, application

Description automatically generated

Figure: LDAA 0, X

Index register X is pointing to m[1000] which contains $A5. Accumulator A will have the value $A5. In binary,

$A5 = 10100101.

Since it is negative, so N = 1. But C, V, Z will be disabled.

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

Graphical user interface, application

Description automatically generated

Figure: ADDA 1,X

X is pointing to m[1001] which has a value $A4. So, A= A+$A4 = $A5+$A4 = $49 since A can contain only 2 byte.

But $A5+$A4 = $149 so it has carry out of 1. Again, signed bit of the result changed for which overflow is 1. So, C and V will be enabled and N, Z will be disabled.

Graphical user interface, application

Description automatically generated

Figure: ADDA 2,X

2,X is referring to m[1002] which has a value of $20. So,

A = A + $20=$49+$20 = $69

Here, C,Z,N,V are all disabled.

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

Graphical user interface, application

Description automatically generated

Figure: ADDA 3,X

3,X is referring to m[1003] which has a value of $97. So,

A = A + $97 = $69+$97 = $00 since A can contain only 2 byte.

But, $69 + $97 = $100. So, carry out is 1 and all bytes are 0. So, C and Z will be enabled. V and Z are disabled.

Graphical user interface, application, Word

Description automatically generated

Figure: STAA $1210

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

The value of A is stored at m[1210]. So, m[1210] = $00

Verification:

LDX #$1000; C, Z, V, N = 0

LDAA 0,X; N = 1, C, Z, V = 0

ADDA 1,X; C, V = 1, Z, N = 0

ADDA 2,X; C, Z, V, N = 0

ADDA 3,X; C,Z = 1, V = 0 , N= 0

all steps are verified.

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

* 1. ***Task 4:***

First we created a new project and renamed to lab3task4. Then we cleared the sample Fibonacci counter program and all the variables.

We compiled the code without any error and the no error dialogue also appeared. We added the following code under the constant declaration section:

ORG $1000;indicates the operation to start from m[1000]

num1: dc.b -$38; m[1000] = $C8

num2: dc.b $A ; m[1001] = $A

num3: dc.b $32; m[1002] = $32

num4: dc.b $4F ; m[1003] = $4F

By using the ORG $1000, all the operations will start from m[1000].

We converted the decimal values to hexadecimal in the prelab as follows:

-56 = -$38 = 11001000 = $C8

10 = $A = 00001010

50 = $32 = 00110010

79 = $4F = 01001111

We declared 4 constants that store -$38, $A, $32, $4F at m[1000], m[1001] , m[1002], m[1003] respectively. Since the memory addresses cannot store a negative value directly, it performs 2’s complement and then stores it.

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

Text

Description automatically generated

Figure: constant section for task 4

Under endless loop, we added the following code:

LDAA num1;A=$C8

SUBA num2;A=$C8-$A=BE

SUBA num3;A=$BE-$32=$8C

ADDA num4;A=$8C+$4F=$DB

STAA $1330; m[1330]=$DB

The overall code looks like this:

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

Text

Description automatically generated

Figure: task 4 source code

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

Graphical user interface, application, Word

Description automatically generated

Figure: LDAA num1

Accumulator A will have the value of memory address pointed by num 1. So, [A] = $C8. Since, $C8 = 11001000 in binary, so N will be enabled as it has negative parity bit. C, V, Z will be disabled.

Graphical user interface, application

Description automatically generated

Figure: SUBA num2

Num2 refers to the value inside m[1001] which is $A.

A = A- $A = $C8 - $A = $BE.

$BE = 10111110 in binary. So, N is 1. Z, V and C = 0

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

Graphical user interface, application, Word

Description automatically generated

Figure: SUBA num3

Num3 refers to the value inside m[1002] which is $32.

A = A- $32 = $BE - $32 = $8C.

$8C = 10001100 in binary. So, N is 1. C,V,Z are disabled.

Graphical user interface, application

Description automatically generated

Figure: ADDA num4

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

Num4 refers to the value inside m[1003] which is $4F.

A = A+$4F = $8C - $4F = $DB.

$DB = 11011011 in binary. So, N is 1. C, V and Z are 0.

Graphical user interface, application, Word

Description automatically generated

Figure: STAA $1330

M[1330] will have the value of accumulator A which is $DB

Verification:

LDAA num1;A=$C8, N=1, C,V,Z=0

SUBA num2;A=$C8-$A=BE, N=1, C,V,Z=0

SUBA num3;A=$BE-$32=$8C, N=1, C,V,Z=0

ADDA num4;A=$8C+$4F=$DB, N=1, C,V,Z=0

All the hand analysis matches with the code output.

Lab 3: S12(x) Assembler Directives and CCR Thursday, 2.30PM

1. **Analysis and Conclusions:**

In this lab, we were able to declare variables and constants to use them as a reference in the source code. By using ORG opcode, we can assign any memory addresses to any number of constants and variables. However, the addresses referred by the variables under variable section can also have a different value than initial value after operations. But the value of memory addresses referred by the constants in constants section cannot be changed without interacting with the constants manually.

We were introduced with the CCR register that indicates the state of the operation after executing each line of code. We only used 4 CCR conditions which are C(carry over), N(negative), V(overflow) and Z(all values are 0). This allowed us to understand the difference between carry over and overflow. When the result cannot be stored in the desired location then it has a carry over. On the other hand, when result of the operations between two same signed bit has a different bit, then it has a overflow. Again, during 2’s complement we can take negative value which can be indicated by N in CCR. When all bytes are 0 then, Z becomes enabled. In the prelab, the concepts of overflow and carry out was considered almost the same. After this lab, the concepts are clarified.