

EEE404/591 – Real Time DSP – Lab 2

Introduction to DSP56800E Family

- *Hemanth Balaji Dandi*
#1213076538

REPORT

Objective

The objective of this lab is to explore the basics of the DSP56800E assembly instruction set.

The following areas will be explored:

- Assembler statement syntax and directives.
- Different addressing modes.
- Arithmetic operations.
- Branch and looping instructions

Example 3

The screenshot shows the IAR Embedded Workbench interface for the file `Idm.elf`. The assembly code window displays the following code:

```
; metrowerks sample c
section rlib
org p:

global Fmain

SUBROUTINE "Fmain",Fmain,FmainEND-Fmain

; assembly level entry point
Fmain:
    MOVE.W #10, A      ;load accumulator A with 16-bit decimal 10
    MOVE.W #%1010, B    ;load accumulator B with 16-bit binary 000000000000001010
    MOVE.L #$A, C       ;load accumulator C with 32-bit hexadecimal $0000000A
    rts

FmainEND:
endsec
```

The registers window shows the following values:

Register	Value
A	0x000000A0000
A0	0x0000
A1	10
A2	0x00
B	0x000000A0000
B0	0x0000
B1	0b00000000000000001010
B2	0x00
C	0x000000000A
C0	0x000A

In what portion of the accumulators (i.e.: A0, A1 or A2) is the number loaded for each instruction? Which instruction should be used to load accumulator A with decimal value 100000? Why?

- **For every instruction, the number is loaded in A1, B1 and C1 respectively, as the number first loads in the MSB region (A1) and then moves onto the LSB region (A0). A2,b2,c2 takes care of the sign bit.**
- **To load the accumulator with the decimal value of 100000, we need Long number of bits, hence, instead of using MOVE.W, we use MOVE.L to load the accumulator. (See figure below)**
- **MOVE.W is to move a Word bit (16-bit) and MOVE.L is to move a Long bit (32-bit)**

The screenshot shows a debugger window with two main panes. The left pane displays assembly code for a file named '1dm.elf'. The code includes directives like .metrowerks sample code, .section rlib, .org p:, .global Fmain, and a subroutine definition for Fmain. It also contains assembly-level entry point instructions: MOVE.L #100000, A; MOVE.W #1010, B; MOVE.L #\$A, C; and rts. The right pane is a 'Registers' window titled '56800E Simulator' showing the state of the 'Thread 0x0' with 'General Purpose Registers'. The registers listed are A, A0, A1, A2, B, B0, and B1. Their values are 100000, 0x86A0, 0x0001, 0x00, 0x000000A0000, 0x0000, and 0x000A respectively.

```

; metrowerks sample code

section rlib
.org p:
.global Fmain

SUBROUTINE "Fmain",Fmain,FmainEND-Fmain

; assembly level entry point
Fmain:
    MOVE.L #100000, A    ;load accumulator A with 32-bit decimal 100000
    MOVE.W #1010, B      ;load accumulator B with 16-bit binary 00000000000000001010
    MOVE.L #$A, C        ;load accumulator C with 32-bit hexadecimal $00000000A
    rts

FmainEND:
    endsec
end

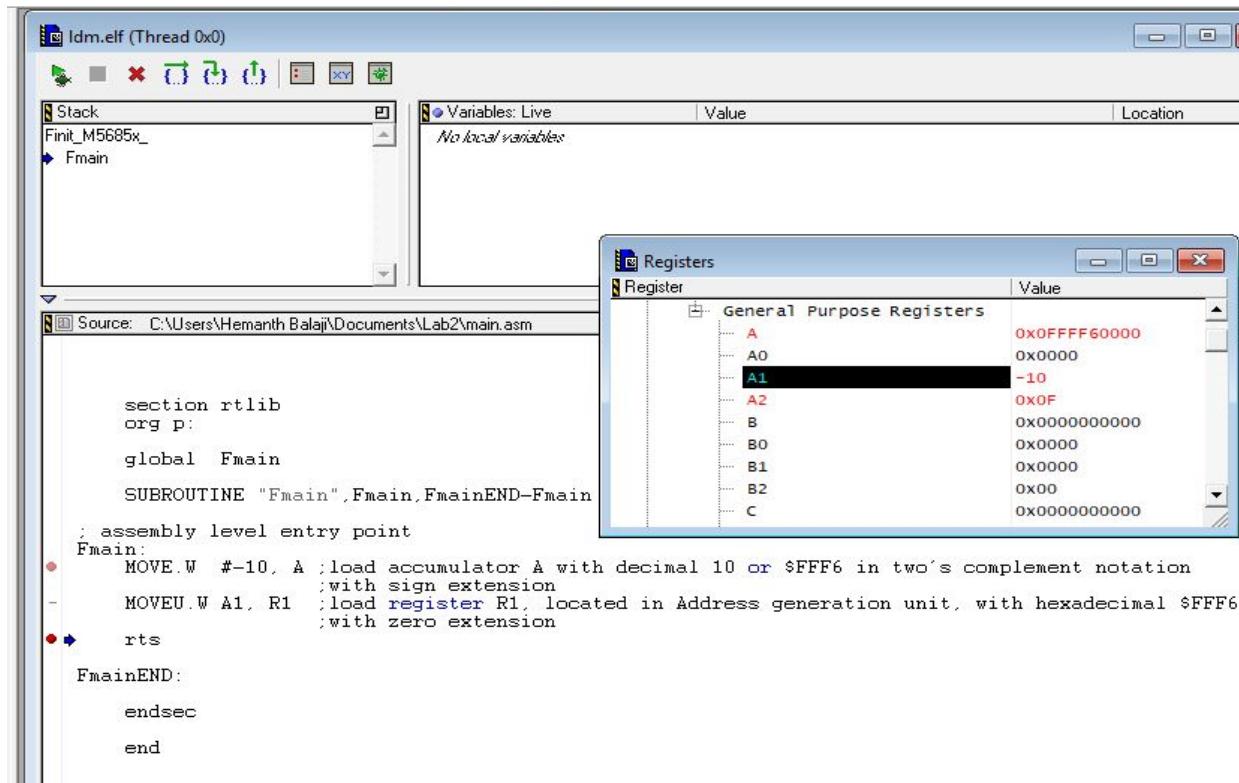
```

Register A1 value when 100000 decimal is moved to A as MOVE.L

Example 4

What is the difference between the output of MOVE.W A1, R1 and MOVEU.W A1, R1?

- ***The MOVEU.W instruction loads A1/R1 with the required value without paying heed to the sign bit, whereas MOVE.W loads the accumulator or register with the required value along with the sign bit. Thus MOVEU.W moves along an unsigned value and MOVE.W move along a signed value.***



Value of Accumulator A

Register	Value
D1	0x0000
D2	0x00
X0	0xFFFF
Y	0x00000000
Y0	0x0000
Y1	0x0000
R0	0x0004BE
R1	0x00FFF6
R2	0x000000
R3	0x000000

Value of Register R1

Example 5

The screenshot shows a debugger interface with several windows:

- Stack:** Shows the current stack frame, which is Finit_M5685x_.
- Variables: Live:** Shows no local variables.
- Registers:** Shows the following register values:

Register	Value
D1	0x0000
D2	0x00
X0	0xFF0A
Y	0x00000000
Y0	0x0000
Y1	0x0000
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x00000000

Source: C:\Users\Hemanth Balaji\Documents\Assembly\metrowerks sample code

```
; metrowerks sample code

section rtlib
org p:

global Fmain

SUBROUTINE "Fmain",Fmain,FmainEND-Fmain

; assembly level entry point
Fmain:
    MOVE.W #$FF0A, X:$0000 ;load data memory $0000 with $FF0A
    MOVE.W #$FF0B, X:$0001 ;load data memory $0001 with $FF0B
    MOVE.L #$0000, R0        ;store $0000 into R0
    MOVE.W X:(R0), X0       ;read data memory $0000 to internal register X0
    rts

FmainEND:
    endsec
end
```

What will be the content of register X0?

- ***The content of X0, as seen from the figure, is FF0A as expected as it is stored in \$0000 address.***

Addressing Modes Used :

- ***The first operand uses immediate value addressing, where a value is stored in an address location directly, while the second operand uses absolute addressing.***
- ***The third instruction uses register direct mode where an internal register is directly referenced as the second operand (R0).***
- ***The fourth instruction uses indirect addressing mode for the first operand, ie, X0 contains the value stored in the R0 which ‘points’ to the address location \$0000, which contains, the value FF0A, thereby acting as a pointer.***
- ***When consecutive memory locations are accessed, it is recommended to use indirect addressing mode.***

Example 6 (To read the value in memory location \$0001 and store it in Y0)

The screenshot shows a debugger interface with the following components:

- Top Bar:** Shows the file "ldm.elf (Thread 0x0)" and various control icons.
- Stack View:** Displays the stack contents, showing symbols like "Init_M5685x_" and "Fmain".
- Variables View:** Shows a table titled "Variables: Live" with the message "No local variables".
- Registers View:** Shows a table titled "Registers" with the following data:

Register	Value
D0	0x0000
D1	0x0000
D2	0x00
X0	0xFF0A
Y	0x0000FF0B
Y0	0xFF0B
Y1	0x0000
R0	0x00000001
R1	0x00000000
R2	0x00000000

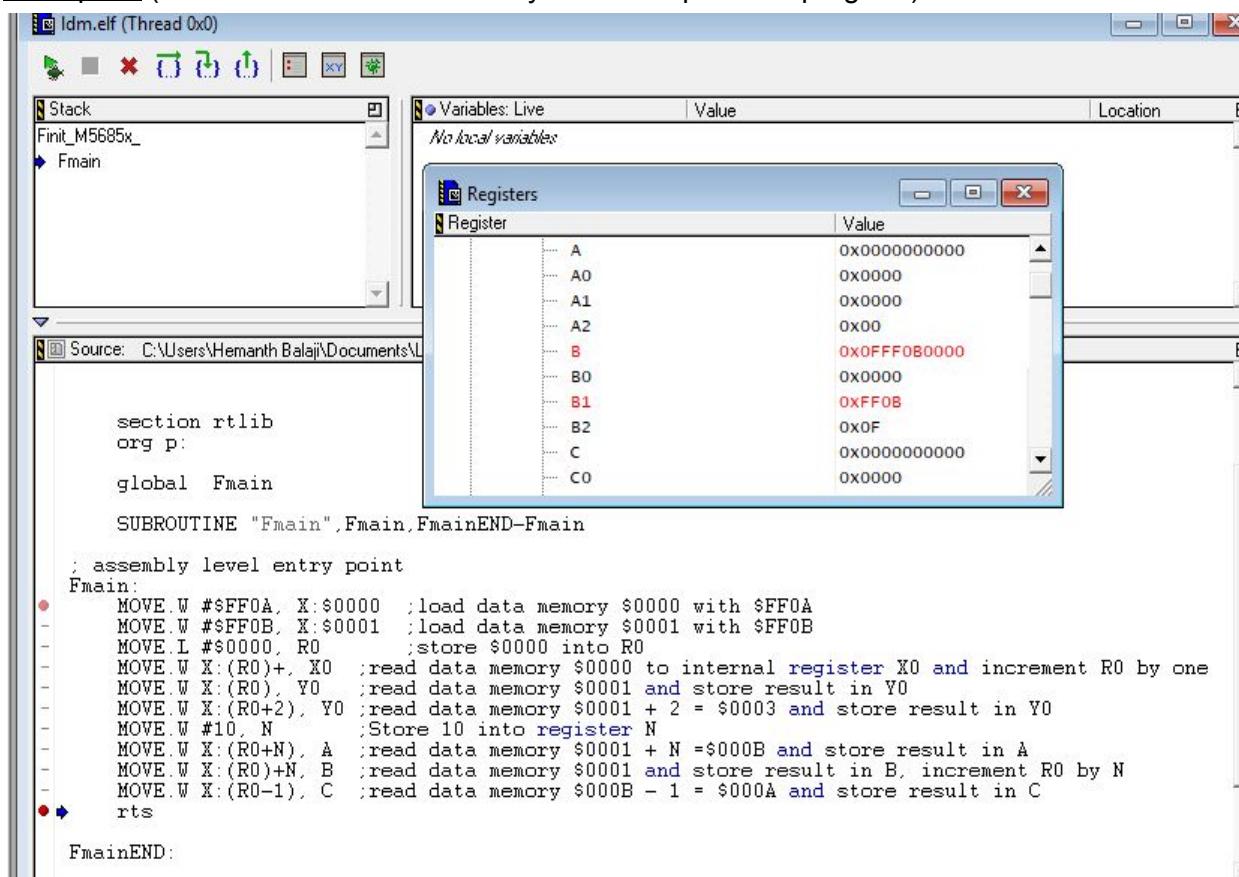
Source Code Area: Displays the assembly code for the "Fmain" subroutine:

```
section rplib
org p:
global Fmain
SUBROUTINE "Fmain",Fmain,FmainEND-Fmain
; assembly level entry point
Fmain:
    MOVE.W #$FF0A, X:$0000 ;load data memory $0000 with $FF0A
    MOVE.W #$FF0B, X:$0001 ;load data memory $0001 with $FF0B
    MOVE.L #$0000, R0 ;store $0000 into R0
    MOVE.W X:(R0), X0 ;read data memory $0000 to internal register X0
    ADDA #1, R0 ;Add 1 to R0 and store the result back in R0
    MOVE.W X:(R0), Y0 ;read data memory $0001 and store result in Y0
    rts

FmainEND:
endsec
```

The assembly code includes comments explaining each instruction's purpose in the context of the example.

Example 7 (To reduce instructions and cycles in the previous program)



The cycles and instruction counts are reduced because

- We replace the

MOVE.W X:(R0),B

ADDA N,R0

by

MOVE.W X: (R0)+N, B

Here, the value in R0 is the memory address and the value at that address is moved into B1. After the move has happened, R0 is incremented by N, and thereby equivalent to the initial two instructions.

Example 8 (To compare the advantages of MOVE.BP over MOVE.B)

Objective : To print out the respective Byte contents of each word, ie, 2,3 then 4,5

Program without using MOVE.BP

The screenshot shows the 56800E Simulator interface with the file 'Idm.elf' open. The assembly code in the source window is as follows:

```
SUBROUTINE "Fmain",Fmain.  
; assembly level entry point  
Fmain:  
    ORG X:  
    X1    DCB    2,3,4,5  
    ORG P:  
    MOVE.L #X1, R1      ;move word pointer to R1  
    MOVE.W #0, A        ;clear A  
    MOVE.W #0, B        ;clear B  
    DO    #4, Loop      ;start looping  
    MOVE.B X:(R1+0),B   ;read byte and store in B  
    ADD   B, A          ;Add A + B -> A  
    ADDA  #1, R1         ;increment R1  
Loop:  
    rts  
FmainEND:  
    endsec  
    end
```

The registers window shows the state of the registers for Thread 0x0:

Register	Value
A	0x00000030000
A0	0x0000
A1	0x0003
A2	0x00
B	0x00000030000
B0	0x0000

First prints 3

The screenshot shows the registers window for Thread 0x0 after the first iteration of the loop. The values are:

Register	Value
A	0x00000080000
A0	0x0000
A1	8
A2	0x00
B	0x00000050000
B0	0x0000
B1	5
B2	0x00

Then skips 2, and Prints 5

The problem with the above code is that, it just prints out one Byte from the respective words, ie, the lower bytes of each word, thereby skipping the other bytes of each word,

altogether, as MOVE.B takes in one byte from each word. Thus it outputs 3 and 5, thereby skipping 2 and 4.

To rectify this, we use @lb(value) and MOVE.BP to move across each Byte in the word and display each value as required.

Example 8 (Program using MOVE.BP)

The screenshot shows the IAR Embedded Workbench IDE interface. On the left, the assembly code for Fmain is displayed:

```
SUBROUTINE "Fmain",Fmain,E
; assembly level entry point
Fmain:
    ORG X:
    X1    DCB    2,3,4,5
    ORG P:
    MOVE.L #@lb(X1), R1      ;move word pointer to R1
    MOVE.W #0, A              ;clear A
    MOVE.W #0, B              ;clear B
    DO    #4,Loop             ;start looping
        MOVE.BP X:(R1+0),B   ;read byte and store in B
        ADD    B, A             ;Add A + B -> A
        ADDA   #1, R1            ;increment R1
    Loop:
        rts
FmainEND:
    endsec
    end
```

On the right, the Registers window shows the state of the registers for Thread 0x0:

Register	Value
A	0x0000030000
A0	0x0000
A1	3
A2	0x00
B	0x0000030000
B0	0x0000
B1	3
B2	0x00

First prints 3

The Registers window shows the state of the registers for Thread 0x0 after the loop has completed:

Register	Value
A	0x0000050000
A0	0x0000
A1	5
A2	0x00
B	0x0000020000
B0	0x0000
B1	2
B2	0x00

Then prints 2

Register	Value
Thread 0x0	
General Purpose Registers	
A	0x00000A0000
A0	0x0000
A1	10
A2	0x00
B	0x0000050000
B0	0x0000
B1	5
B2	0x00

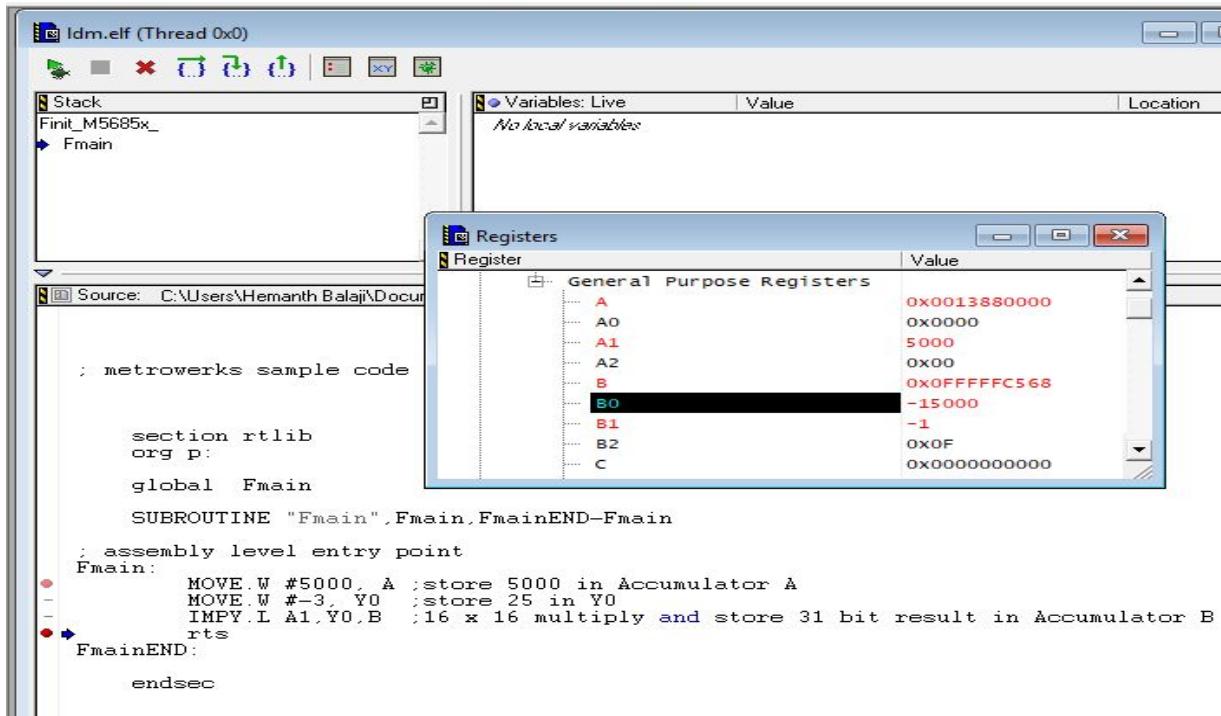
Then Prints 5

Register	Value
Thread 0x0	
General Purpose Registers	
A	0x00000E0000
A0	0x0000
A1	14
A2	0x00
B	0x0000040000
B0	0x0000
B1	4
B2	0x00

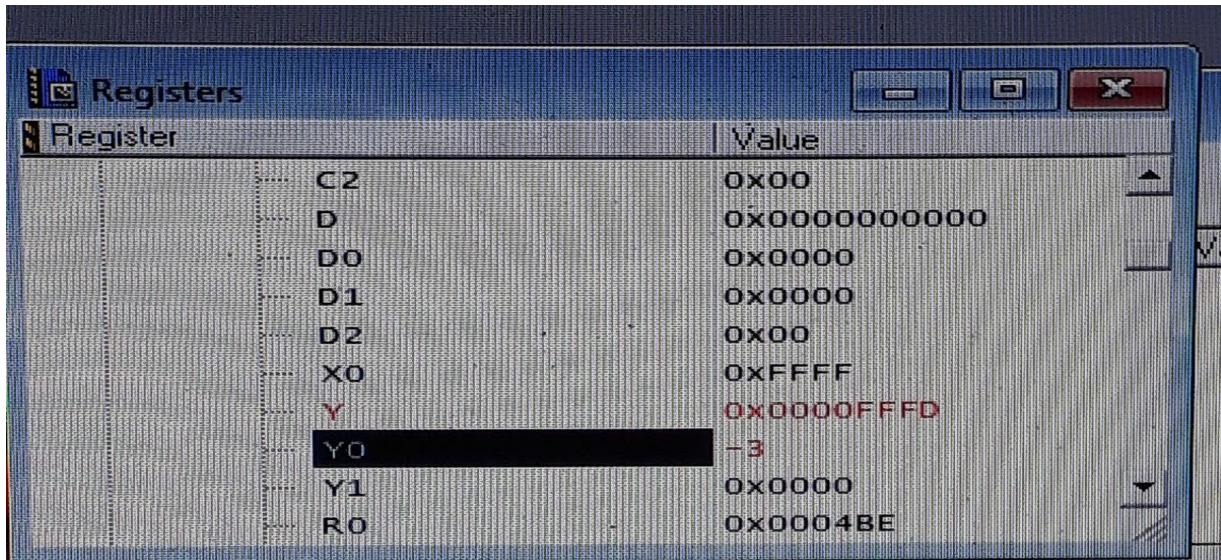
Then Prints 4

Therefore, all the byte values present in the given two words have been printed out, and the advantage of MOVE.BP over MOVE.B has been highlighted.

Example 9



Value stored in A1



Value stored in Y0

In the CW menu select, View > Registers and check for A2, A1, A0, Y0, B2, B1, B0. Where is the multiplication result located? By default, results are shown in hexadecimal; right click on the hexadecimal number and select —View as Signed Decimal || to change to decimal.

- The values for A2, A1, A0, Y0, B2, B1, B0 have all as given in the above images and below image. The result is stored in the B register, specifically B1. The equivalent signed decimal values of the default hexadecimal values are shown in the 'Registers' window.

Example 10

The screenshot shows the IAR Embedded Workbench interface with the project 'ldm.elf' open. The assembly code for the function Fmain is displayed in the left pane:

```

; metrowerks sample code

section rlib
org p:

global Fmain

SUBROUTINE "Fmain",Fmain,FmainEND-Fmain

; assembly level entry point
Fmain:
    MOVE.W #5000, A ;store 5000 in Accumulator A
    MOVE.W #25, Y0 ;store 25 in Y0
    IMPY.L A1,Y0,B ;16 x 16 multiply and store 31 bit result in Accumulator B
    rts
FmainEND:
    endsec
    end

```

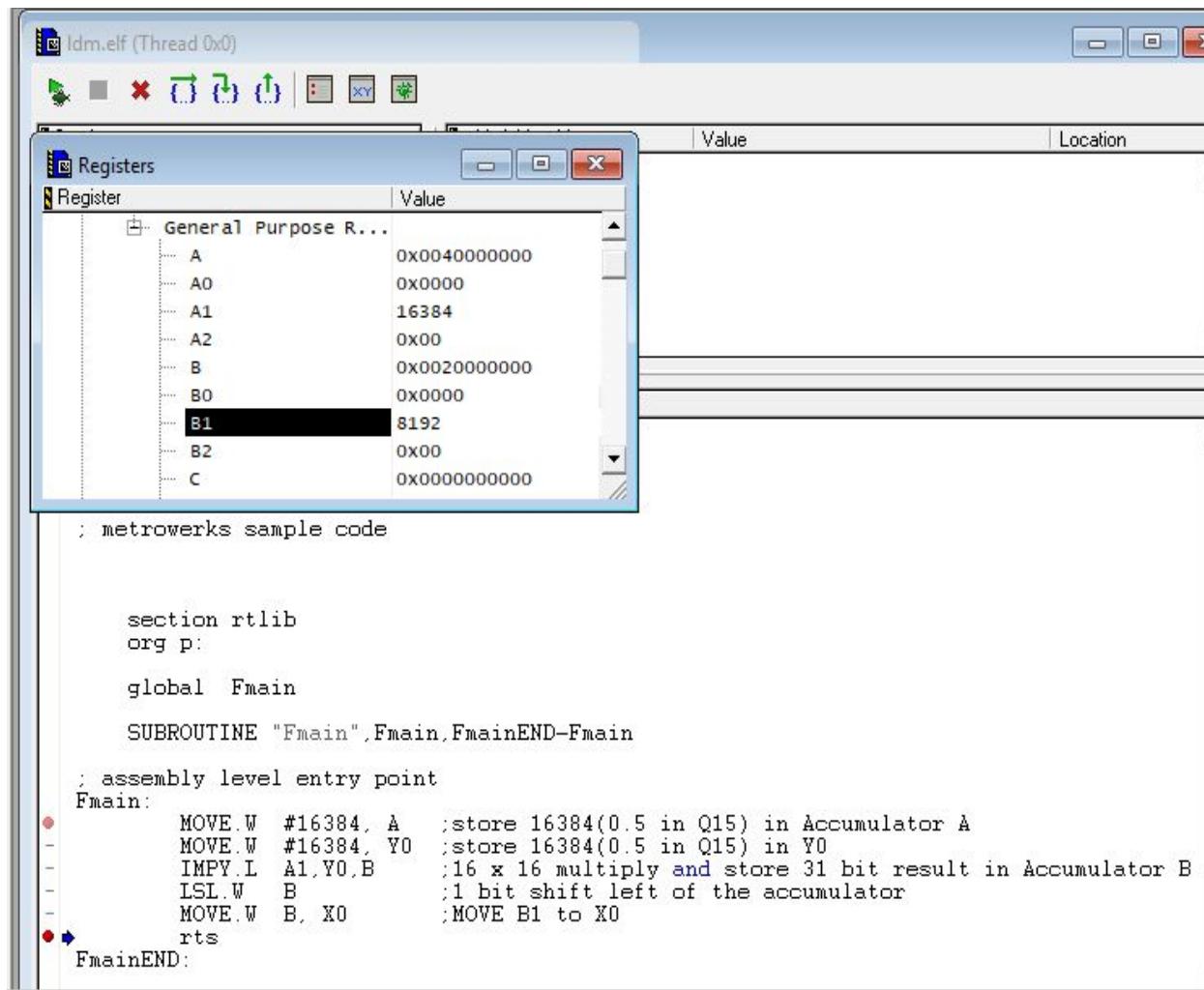
The assembly code includes instructions to move immediate values into registers A and Y0, and then perform a 16x16 multiplication (IMPY.L) with the result stored in register B. The 'Registers' window (bottom right) shows the following values:

Register	Value
A	0x0013880000
A0	0x0000
A1	5000
A2	0x00
B	125000 (highlighted)
B0	-6072
B1	1
B2	0x00
C	0x000000000000

Check the accumulator B for the result, is overflow occurring? Usually, the result needs to be stored back in memory and, in this case, the result will occupy two memory locations. This is not recommended for large algorithms. Can you think of a mechanism to prevent overflow using only 16-bit results?

- In the above example, in-order to print out a number whose range lies outside of the 32,767 and -32,768 range, we use Long multiplication to accommodate 12500, which is greater than 32,767. If we look at the B register, we notice because two memory locations are used, overflow has occurred.
- In-order to prevent overflow, we use Fractional multiplication using MPY which keeps the 16-bit results by division by a scaling factor to keep it normalized/quantized between 0 and 1.

Example 11- Fractional Multiplication



Fractional Multiplication b/w 2 16-bit numbers

Check the values in B1, B0 and X0. What is the result of the multiplication?

- ***The values for B1,B0 and X0 are given in the above figure.***

Fractional vs Decimal Multiplication

- ***If we multiply two 16-bit numbers, from decimal multiplication, we expect a 32-bit number***
- ***If we multiply the same two 16-bit numbers by Fractional Multiplication, we will get a 16-bit number, therefore, maintain the memory of the chip.***

Example 12-To save cycles, we use MPY for Fractional Multiplication

The screenshot shows the IAR Embedded Workbench interface. The assembly code in the editor window is:

```

section rtlib
org p:

global Fmain

SUBROUTINE "Fmain",Fmain,FmainEND-Fmain
; assembly level entry point
Fmain:
    MOVE.W #16384, A      ;store 16384(0.5 in Q15) in Accumulator A
    MOVE.W #16384, Y0     ;store 16384(0.5 in Q15) in Y0
    MPY A1,Y0,B           ;16 x 16 fractional multiply
    MOVE.W B, X0           ;MOVE B1 to X0
rts
FmainEND:
endsec

```

The Registers window shows the following values:

Register	Value
A	0x0040000000
A0	0x0000
A1	16384
A2	0x00
B	0x0020000000
B0	0x0000
B1	8192
B2	0x00
C	0x000000000000

Check again for the result in B1, B0 and X0. Is it the same as the one previously obtained?

- Yes, the values are exactly the same.

Fractional Multiplication b/w 5/2¹⁵ and 5/2¹⁵

The screenshot shows the IAR Embedded Workbench interface. The assembly code in the editor window is:

```

; metrowerks sample

section rtlib
org p:

global Fmain

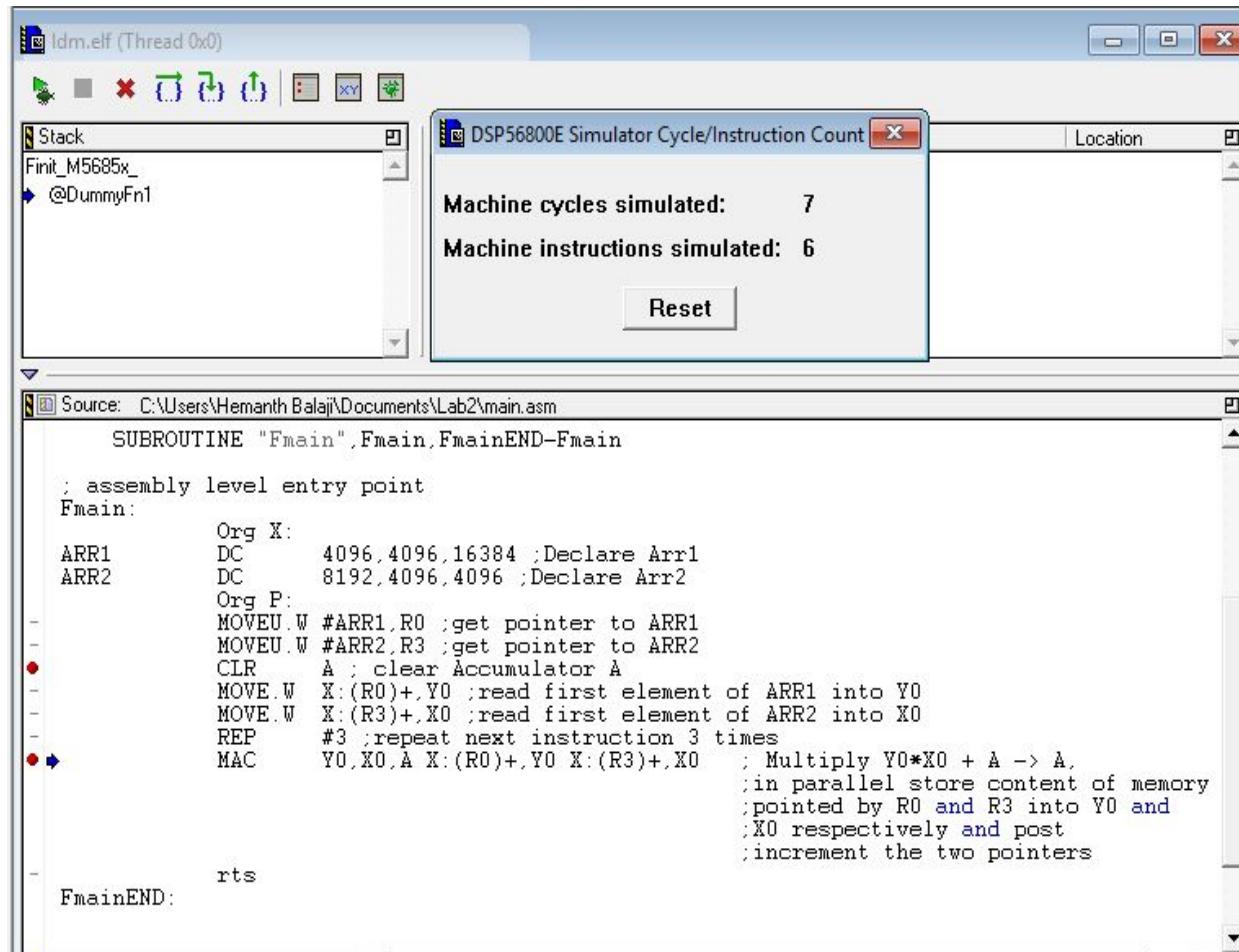
SUBROUTINE "Fmain",Fmain,FmainEND-Fmain
; assembly level entry point
Fmain:
    MOVE.W #5, A      ;store 5 in Accumulator A
    MOVE.W #5, Y0     ;store 5 in Y0
    MPY A1,Y0,B       ;16 x 16 fractional multiply
    MOVE.W B, X0       ;MOVE B1 to X0
rts
FmainEND:
endsec
end

```

The Registers window shows the following values:

Register	Value
A	0x00000050000
A0	0x0000
A1	5
A2	0x00
B	0x00000000032
B0	50
B1	0x0000
B2	0x00
C	0x000000000000

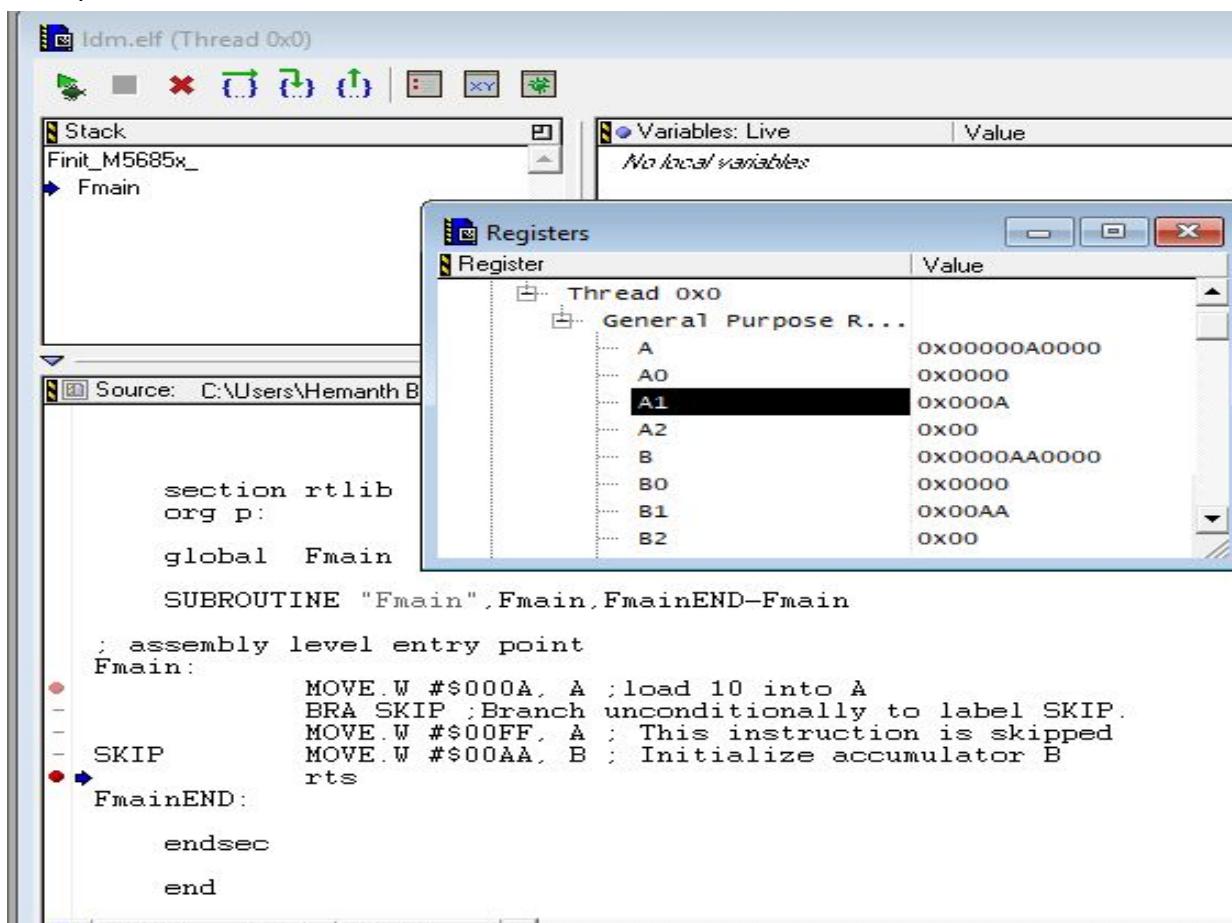
Example 13 (MAC Operation- Multiply and Add Operation)



Measure the number of cycles and instruction count for the MAC operation. (Start measuring at the instruction after the CLR A and upto and including the MAC instruction). Compare with the number of cycles and instruction count you obtained for doing a MAC operation in assembly in PART B of lab 2. (Note: Here you are performing the MAC three times, hence you must multiply the number of cycles and instruction count you obtained from lab2 by three)

	MAC Operation in Lab 1	MAC Operation here
Machine Cycles simulated	$197 \times 3 = 591$	7
Machine Instruction count	$101 \times 3 = 303$	6

Example 14

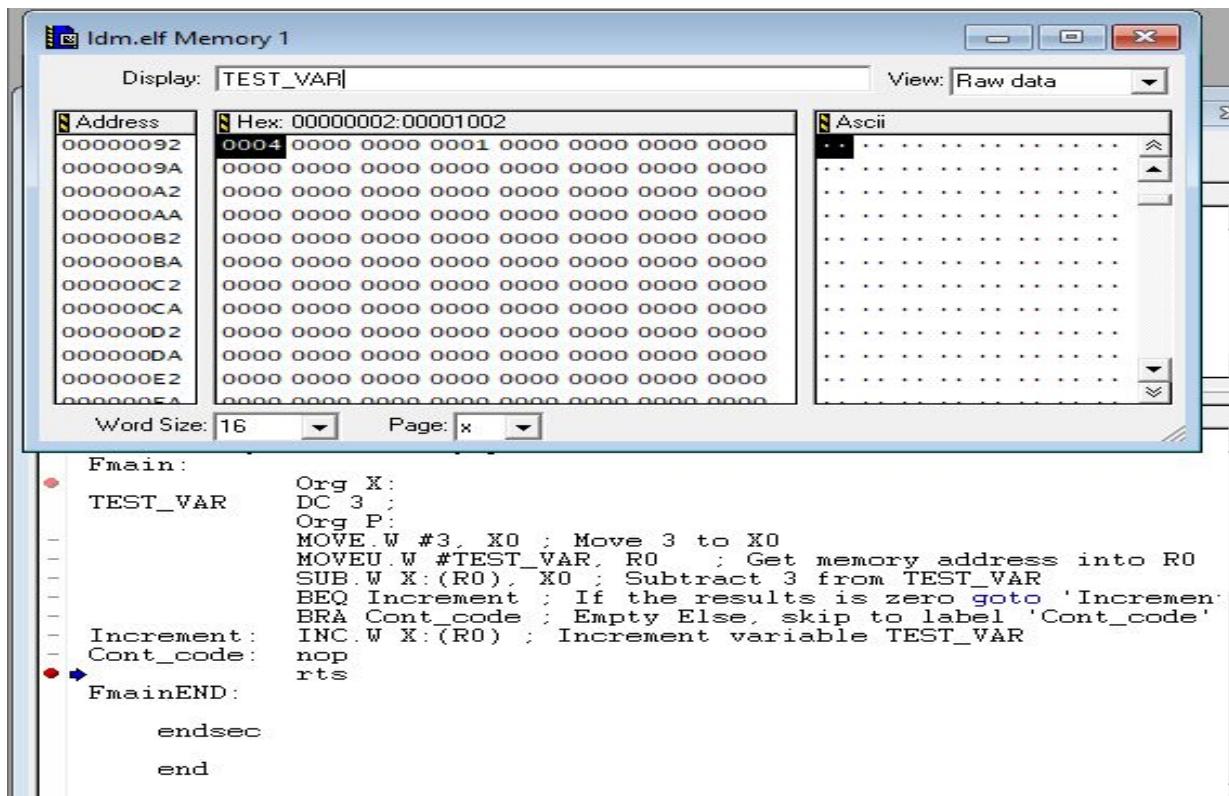


Unconditional branching

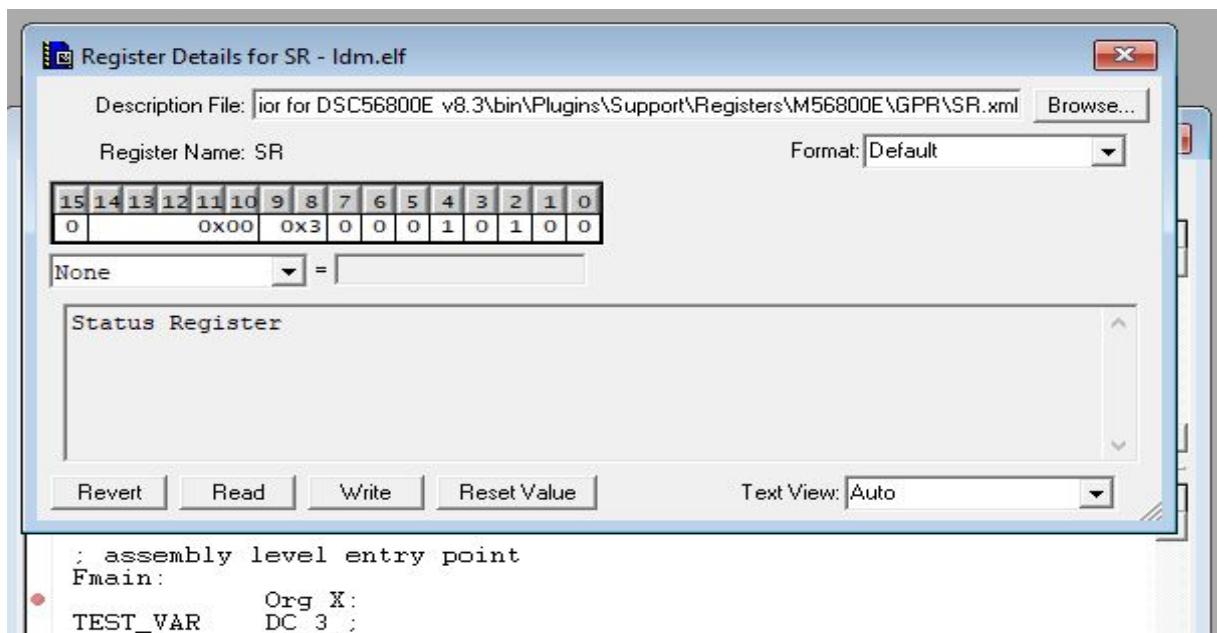
What is the content of accumulator A after code execution?

- **As seen from the above figure, the accumulator has A1-000A**

Example 15 Conditional Branching



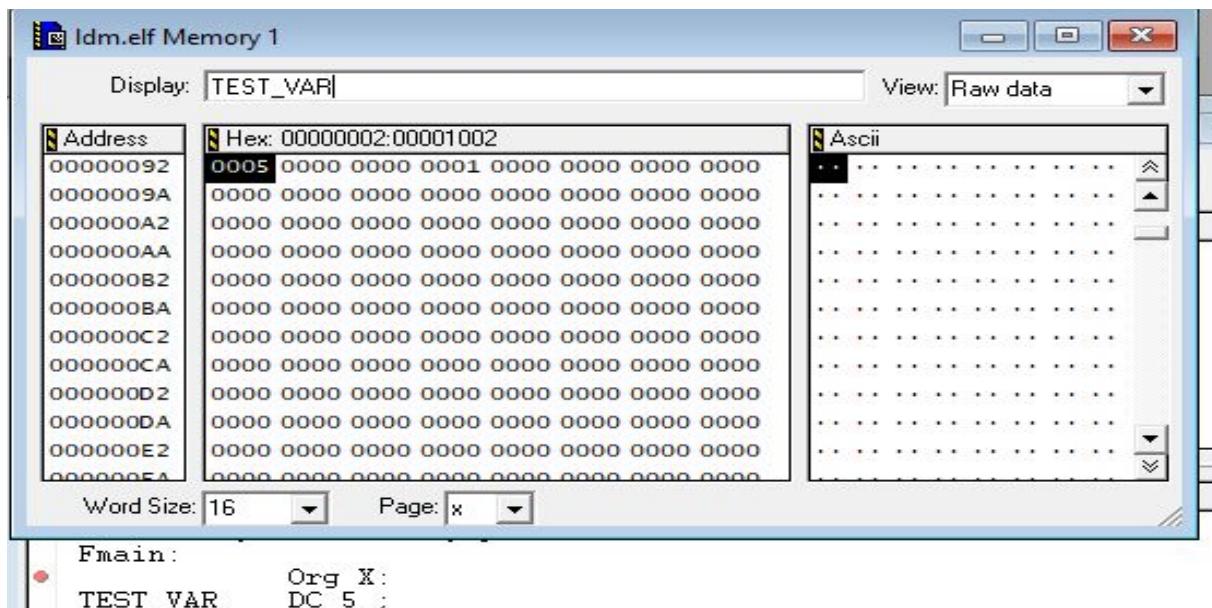
TEST_VAR Final Value



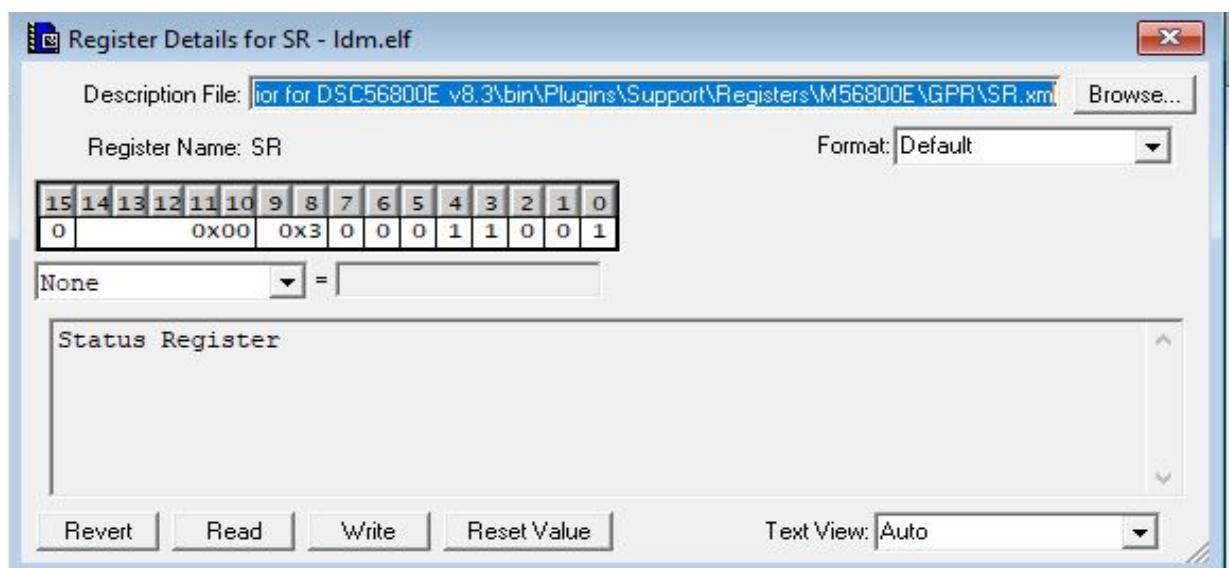
Checking the Status Register for the value of the Z bit.

Is the _BEQ' condition satisfied? Continue executing the program and check the content of the memory location TEST_VAR. Rather than initializing the TEST_VAR to 3 change it to be 5, execute the code again and check for the results.

- **If the last operation result is zero, the the Z bit of the Status register (Bit-2) will be 1 else 0. In this case, the condition was satisfied as the operation before the BEQ statement is zero, hence making the Z bit of the SR as 1, as can be seen from the above figure.**
- **However, if we change TEST_VAR to 5, then when we subtract it from 3, it wouldn't be zero, hence the Z bit will be 0 as shown in the below figure.**



TEST_VAR Final Value.



The Bit-2 of SR is 0 as expected

Using BNE (Grad Only)

Objective: To calculate {if($x==3$) in Assembly using BNE}

```
    x++;  
else  
    x--;}
```

The screenshot shows the IAR Embedded Workbench interface. On the left, the assembly code for the Fmain function is displayed. The code initializes a variable TEST_VAR to 3, then enters a loop where it checks if TEST_VAR is zero using a BNE (Branch if Not Equal) instruction. If TEST_VAR is not zero, it decrements TEST_VAR by 1. If TEST_VAR is zero, it increments TEST_VAR by 1 and skips the next instruction. The code ends with an rts (Return From Subroutine) instruction. On the right, the 'Memory' window titled 'Idm.elf Memory 1' shows the memory dump for the TEST_VAR variable. The address 00000002 is highlighted, showing the value 00000003 (hex 00000003). The memory dump table has columns for Address, Hex, and Ascii. The hex value at address 00000002 is 00000003, and the ASCII representation is three zeros followed by a null terminator.

Address	Hex: 00000002:000001002	Ascii
00000002	0000 0000 0001 0000 0000 0000 0000 0000
0000000A	0000 0000 0000 0000 0000 0000 0000 0000
000000A2	0000 0000 0000 0000 0000 0000 0000 0000
000000AA	0000 0000 0000 0000 0000 0000 0000 0000
000000B2	0000 0000 0000 0000 0000 0000 0000 0000
000000BA	0000 0000 0000 0000 0000 0000 0000 0000
000000C2	0000 0000 0000 0000 0000 0000 0000 0000
000000CA	0000 0000 0000 0000 0000 0000 0000 0000
000000D2	0000 0000 0000 0000 0000 0000 0000 0000
000000DA	0000 0000 0000 0000 0000 0000 0000 0000
000000E2	0000 0000 0000 0000 0000 0000 0000 0000
000000F4	0000 0000 0000 0000 0000 0000 0000 0000

Word Size: 16 Page: x

```
Fmain:  
    .ORG X:  
TEST_VAR    DC      3  
.ORG P:  
    MOVE.W #3,X0          ;Move 3 to X0  
    MOVEU.W #TEST_VAR, R0 ;Get memory address into R0  
    SUB.W  X:(R0),X0       ;Subtract 3 from TEST_VAR  
    BNE    Decrement      ;If the results is zero goto 'Decrement'  
    BRA    Increment      ;Else, go to label 'Increment'  
Increment: INC.W  X:(R0) ;Increment variable TEST_VAR  
Decrement: BRA    SKIP   ;If 'Increment', skip the next instruction  
SKIP     DEC.W  X:(R0) ;Decrement variable 'TEST_VAR'  
FmainEND:  
    endsec  
    end
```

Example 17

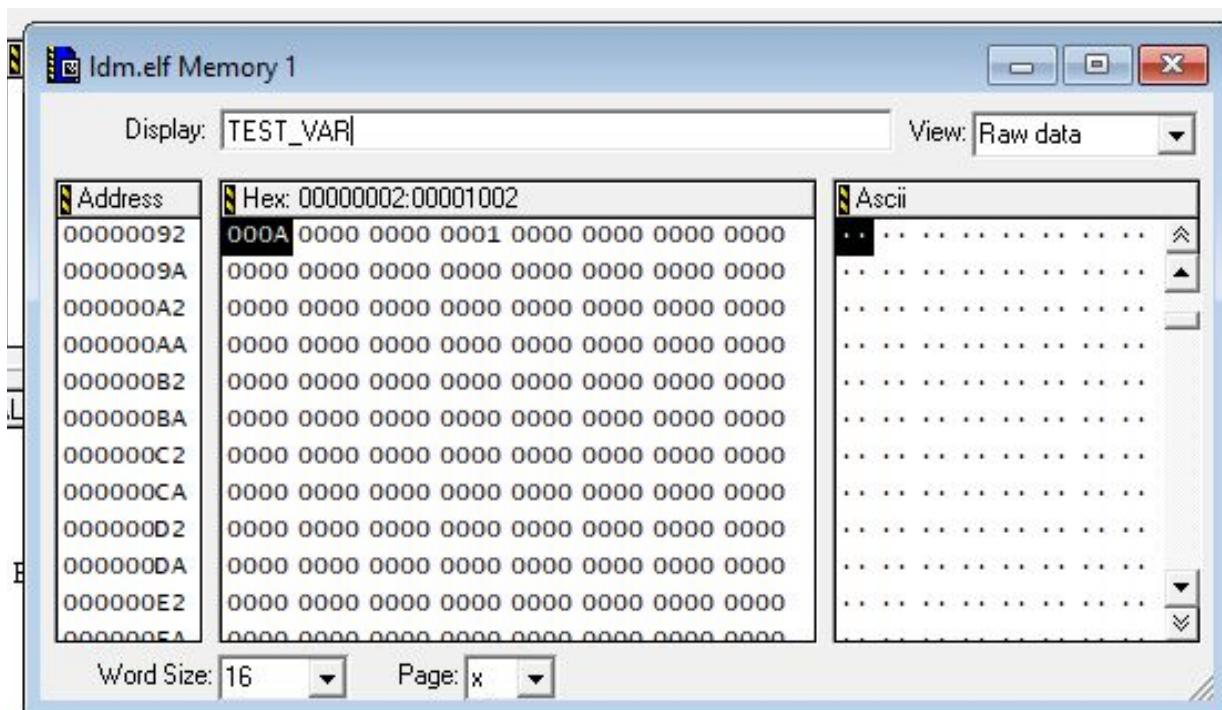
The screenshot shows the assembly code for a program named Fmain. The code includes a subroutine Fmain, a variable TEST_VAR initialized to 0, and a loop that increments TEST_VAR by 1 until it reaches 10. The assembly code is as follows:

```
org p:  
  
global Fmain  
  
SUBROUTINE "Fmain",Fmain,FmainEND-Fmain  
  
; assembly level entry point  
Fmain:  
    ORG X:  
TEST_VAR    DC 0  
    ORG P:  
    MOVEU.W #TEST_VAR, R0 ; Get memory address  
    MOVE.W #10, A1 ; Move 10 to Accumulator  
Loop:    INC.W X:(R0) ; Increment variable TEST_VAR  
        DEC.W A ; Decrement Accumulator by one  
        BNE Loop ; Loop as long as A is different from zero  
        rts  
  
FmainEND:  
  
endsec  
  
end
```

A separate window titled "DSP56800E Simulator Cycle/Instruction Count" displays the results of the simulation:

- Machine cycles simulated: 136
- Machine instructions simulated: 60
- Reset button

Machine Cycles and Instruction Count



Value of TEST_VAR

Example 18

The screenshot shows a software interface for a DSP56800E simulator. On the left, there is a code editor window displaying assembly code. The code starts with a comment ': metrowerks sample code' and defines a section 'rtlib' at address 'p'. It contains a global variable 'Fmain' and a subroutine 'Fmain' from 'Fmain' to 'FmainEND'. Inside 'Fmain', there is an assembly level entry point 'Fmain:' followed by a label 'TEST_VAR'. The code then enters a loop labeled 'Loop:' which repeats 10 times, incrementing the value of 'TEST_VAR' (R0) by 1 each iteration, and ends with a 'rts' instruction. A red dot marks the current instruction being executed. On the right, a separate window titled 'DSP56800E Simulator Cycle/Instruction Count' displays the results of the simulation: 'Machine cycles simulated: 76' and 'Machine instructions simulated: 38'. A 'Reset' button is also present in this window.

```

; metrowerks sample code

section rlib
org p:

global Fmain

SUBROUTINE "Fmain",Fmain,FmainEND-Fmain

; assembly level entry point
Fmain:
    TEST_VAR      ORG X:
    DC 0
    ORG P:
    MOVEU.W #TEST_VAR,R0      ;Get memory address
    Loop:      REP #10          ;Repeat the following instruction 10 times
    INC.W X:(R0)           ;Increment variable TEST_VAR
    rts

```

Machine Cycles and Instruction Count

The screenshot shows a memory dump window titled 'Idm.elf Memory 1'. The 'Display' field is set to 'TEST_VAR'. The window has two main panes: 'Hex' and 'Ascii'. The Hex pane shows memory starting at address 00000092, where the value is 00000002:00001002. The Ascii pane shows the character representation of this hex value, which is a single carriage return character (ASCII 0D). The bottom of the window includes controls for 'Word Size' (set to 16) and 'Page' (set to x).

Address	Hex	Ascii
00000092	0000 0000 0001 0000 0000 0000 0000	\r
0000009A	0000 0000 0000 0000 0000 0000 0000	
000000A2	0000 0000 0000 0000 0000 0000 0000	
000000AA	0000 0000 0000 0000 0000 0000 0000	
000000B2	0000 0000 0000 0000 0000 0000 0000	
000000BA	0000 0000 0000 0000 0000 0000 0000	
000000C2	0000 0000 0000 0000 0000 0000 0000	
000000CA	0000 0000 0000 0000 0000 0000 0000	
000000D2	0000 0000 0000 0000 0000 0000 0000	
000000DA	0000 0000 0000 0000 0000 0000 0000	
000000E2	0000 0000 0000 0000 0000 0000 0000	
000000FA	0000 0000 0000 0000 0000 0000 0000	

Value of TEST_VAR

(Grad Only)

As we can clearly see, the Hardware looping instructions are more efficient than those of software.

Software Looping vs Hardware Looping

- As we can see from Examples 17 & 18, by Hardware Looping, the Machine simulates less number of Instructions and Cycles as when compared to looping performed by software instructions
- Hardware Instructions are performed by two operators- the REP and DO commands. The former can only execute the instruction just after the command number of times, and if we want to execute a block of instructions, we use the latter command.

Example 19

The screenshot shows the IAR Embedded Workbench interface with the project "Idm.elf" open. The assembly code window displays the following code:

```

global Fmain

SUBROUTINE "Fmain",Fmain,Fn1
; assembly level entry point
Fmain:
TEST_VAR    ORG X:
            DC 0
            ORG P:
            MOVEU.W #TEST_VAR, R0      ;Get memory address
            DO #10,Loop               ;Do Looping 10 times
            INC.W X:(R0)              ;Increment variable TEST_VAR
            MOVE.W X:(R0), X0          ;Move TEST_VAR to register X0
            ASL.W X0                  ;Multiply X0 by 2 by shifting to the left
            MOVE.W X0, X:(R0)          ;Store the result back to TEST_VAR
Loop:
            rts

FmainEND:
endsec
end

```

The memory dump window titled "Idm.elf Memory 1" shows the memory starting at address 00000092. The hex dump shows the value 07FE followed by 10 zeros. The ASCII dump shows the character 'A' followed by 10 blank spaces.

Address	Hex	ASCII
00000092	07FE 0000 0000 0001 0000 0000 0000 0000	A.....
0000009A	0000 0000 0000 0000 0000 0000 0000 0000
000000A2	0000 0000 0000 0000 0000 0000 0000 0000
000000AA	0000 0000 0000 0000 0000 0000 0000 0000
000000B2	0000 0000 0000 0000 0000 0000 0000 0000
000000BA	0000 0000 0000 0000 0000 0000 0000 0000
000000C2	0000 0000 0000 0000 0000 0000 0000 0000
000000CA	0000 0000 0000 0000 0000 0000 0000 0000
000000D2	0000 0000 0000 0000 0000 0000 0000 0000
000000DA	0000 0000 0000 0000 0000 0000 0000 0000
000000E2	0000 0000 0000 0000 0000 0000 0000 0000
000000FA	0000 0000 0000 0000 0000 0000 0000 0000

Lab Evaluation:

Difficulties encountered:

- Understanding fractional and decimal multiplications' logic
- Understanding how and why to use MOVE.BP with @lb(value) instead of MOVE.B
- Understanding various addressing modes
- Understanding that BNE is Branch NOT encountered, therefore, is activated when the previous statement is false, not true, unlike BEQ.

Errors or Unclear Statements in Lab Manual:

- Understanding the addressing mode types in Example 5 were a little bit tricky, but fine.
- Every objective was clearly explained

Suggestions:

Level of problems given were fine considering that we have just begun understanding assembly language.

Conclusion

Learnt about

- Addressing Modes
- Branching and Looping Functions
- How to declare variables and where to declare them (Like Test_Var declared in data memory as ORG X and not program memory ORG P)
- Difference between MPY and IMPY
- Difference between MOVE.B and MOVE.BP

Liked Most?

Branching and Looping instructions

Liked Least?

Writing this report (Just kidding!) Not much NOT to like. Assembly language is quite interesting!