EEE404/591 – Real Time DSP – Lab 2
Introduction to DSP56800E Family

- ***Hemanth Balaji Dandi***
#1213076538

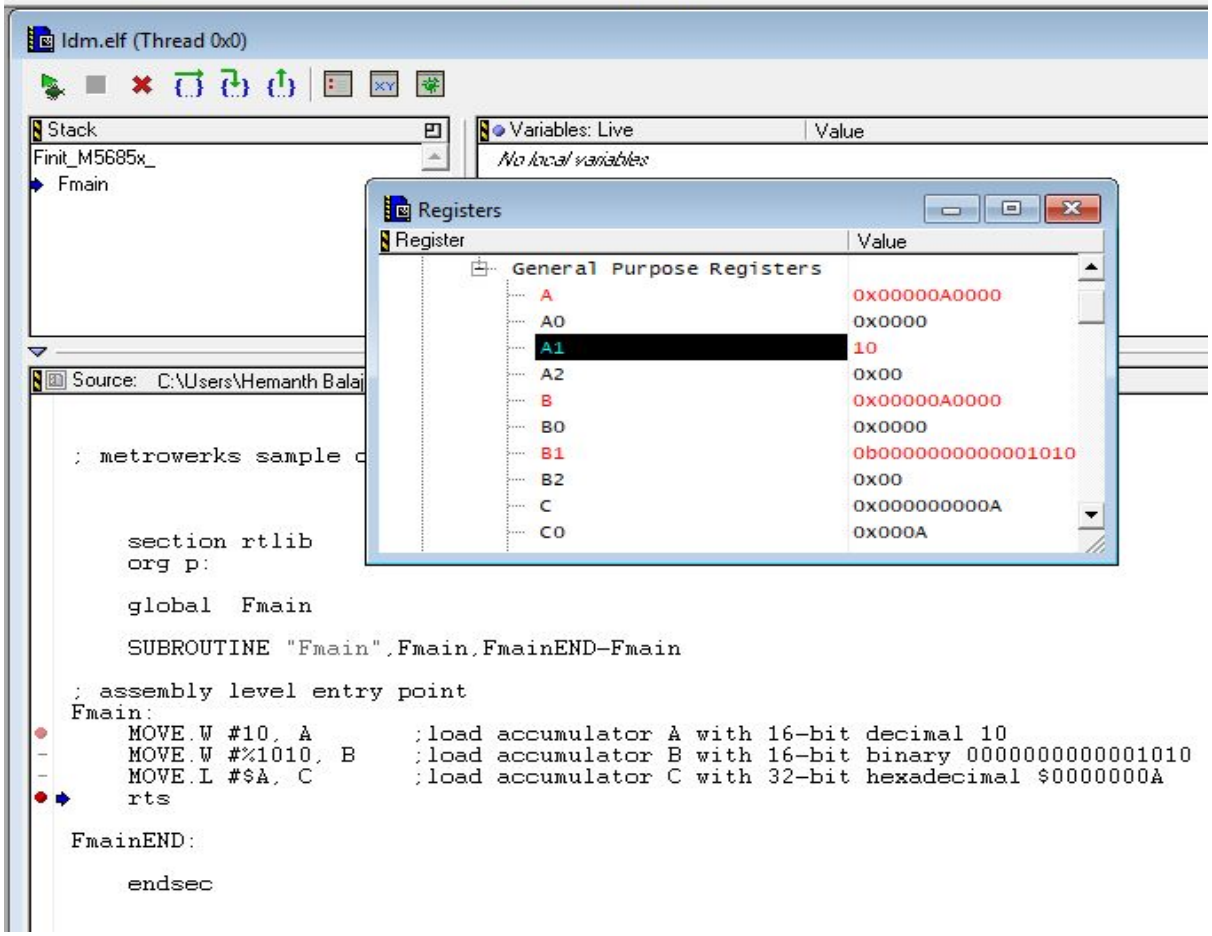
REPORT

Objective

The objective of this lab is to explore the basics of the DSP56800E assembly instruction set. The following areas will be explored:

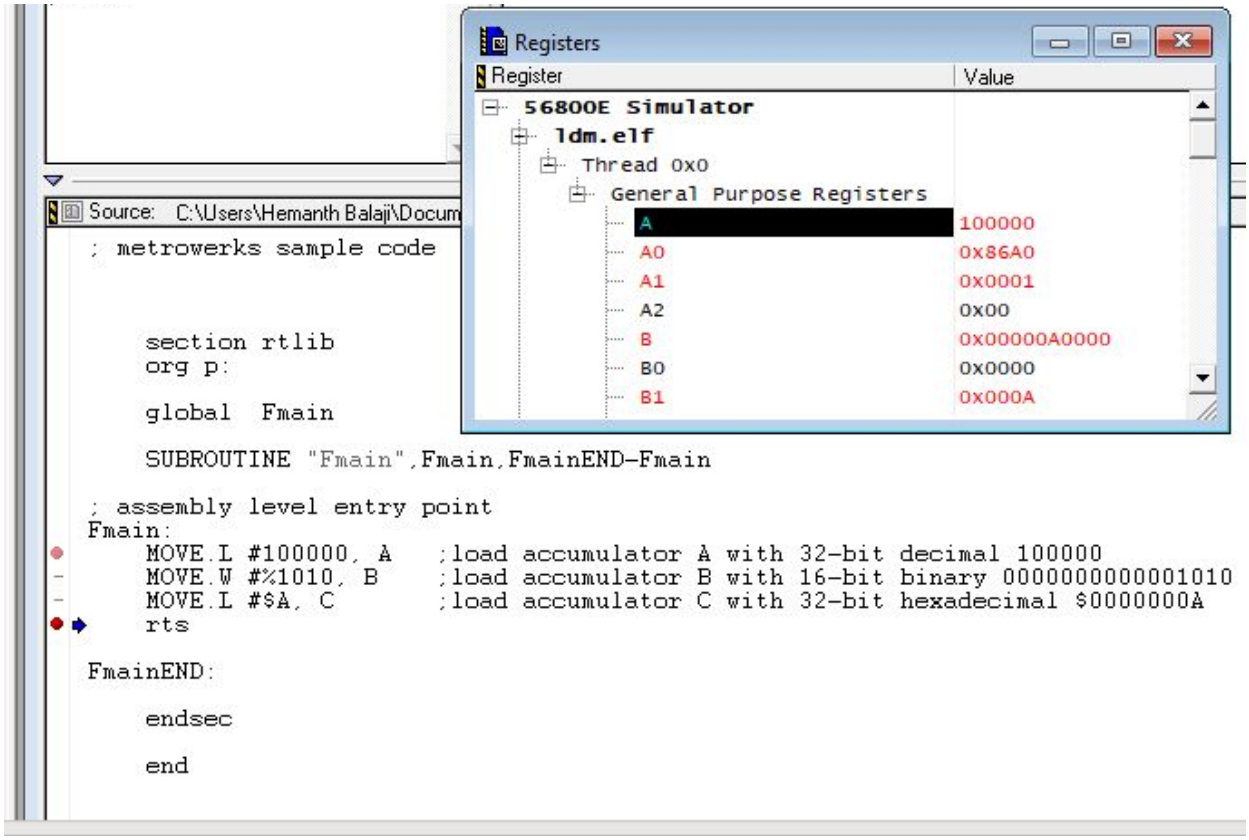
- Assembler statement syntax and directives.
- Different addressing modes.
- Arithmetic operations.
- Branch and looping instructions

Example 3



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)***

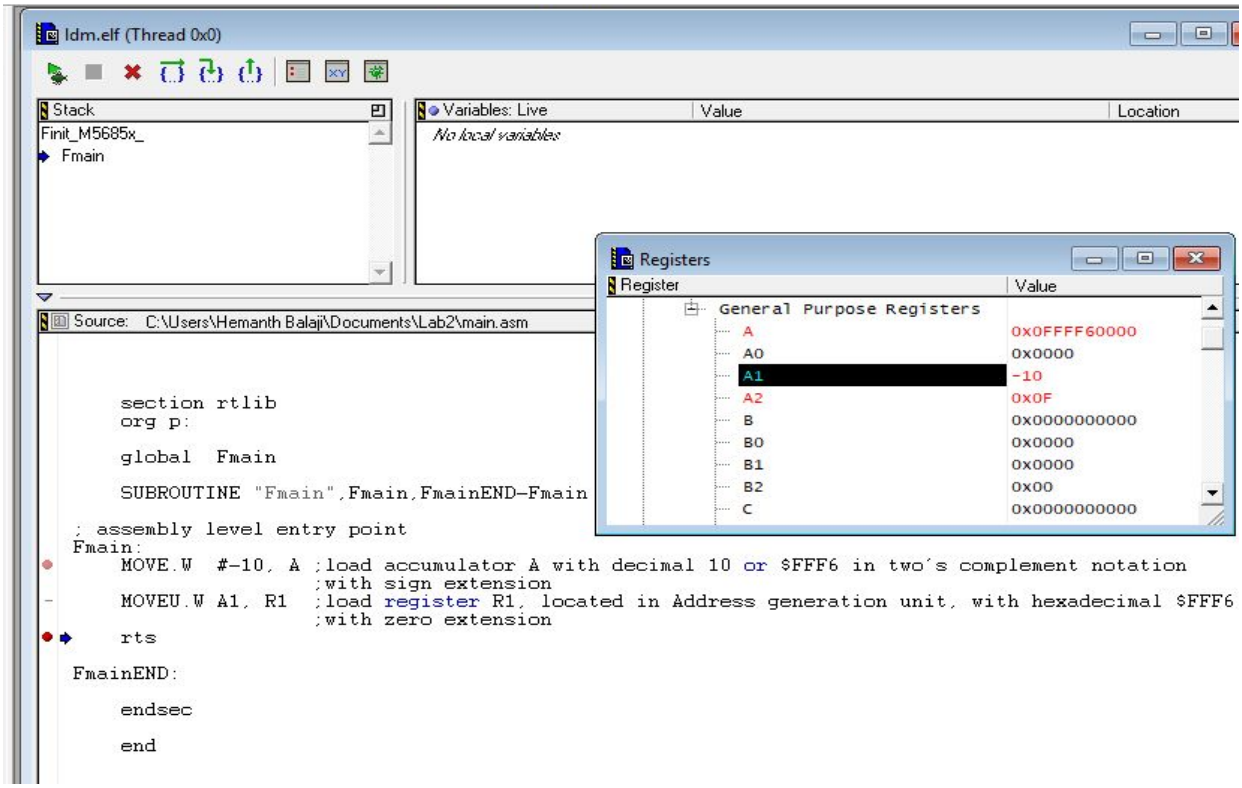


Register A1 value when 100000 decimal is moved to A as MOVE.L

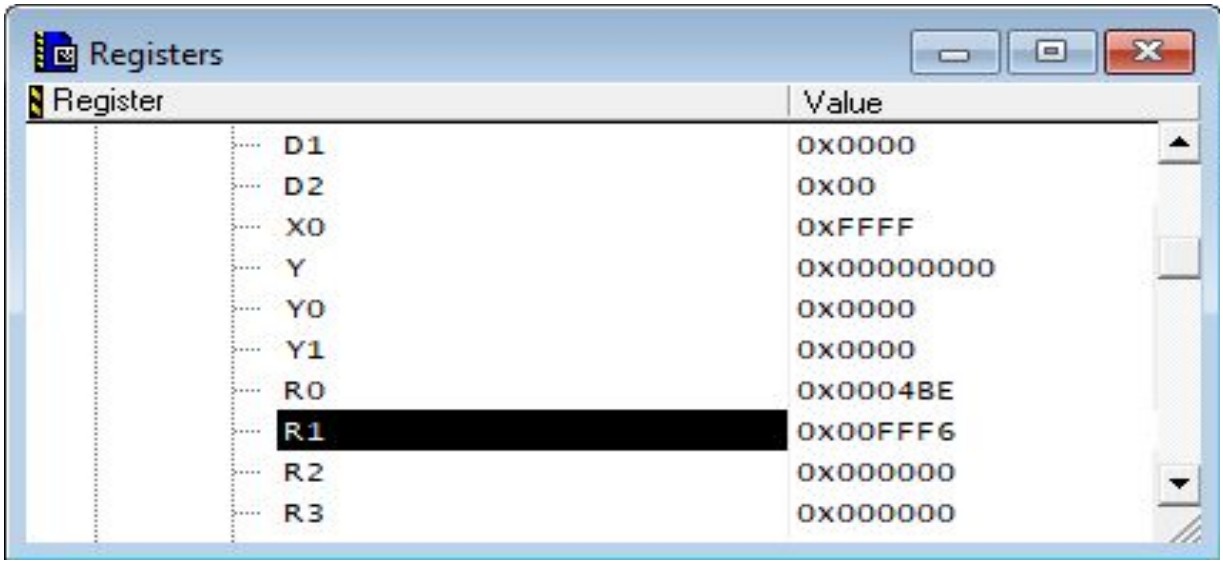
Example 4

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.***

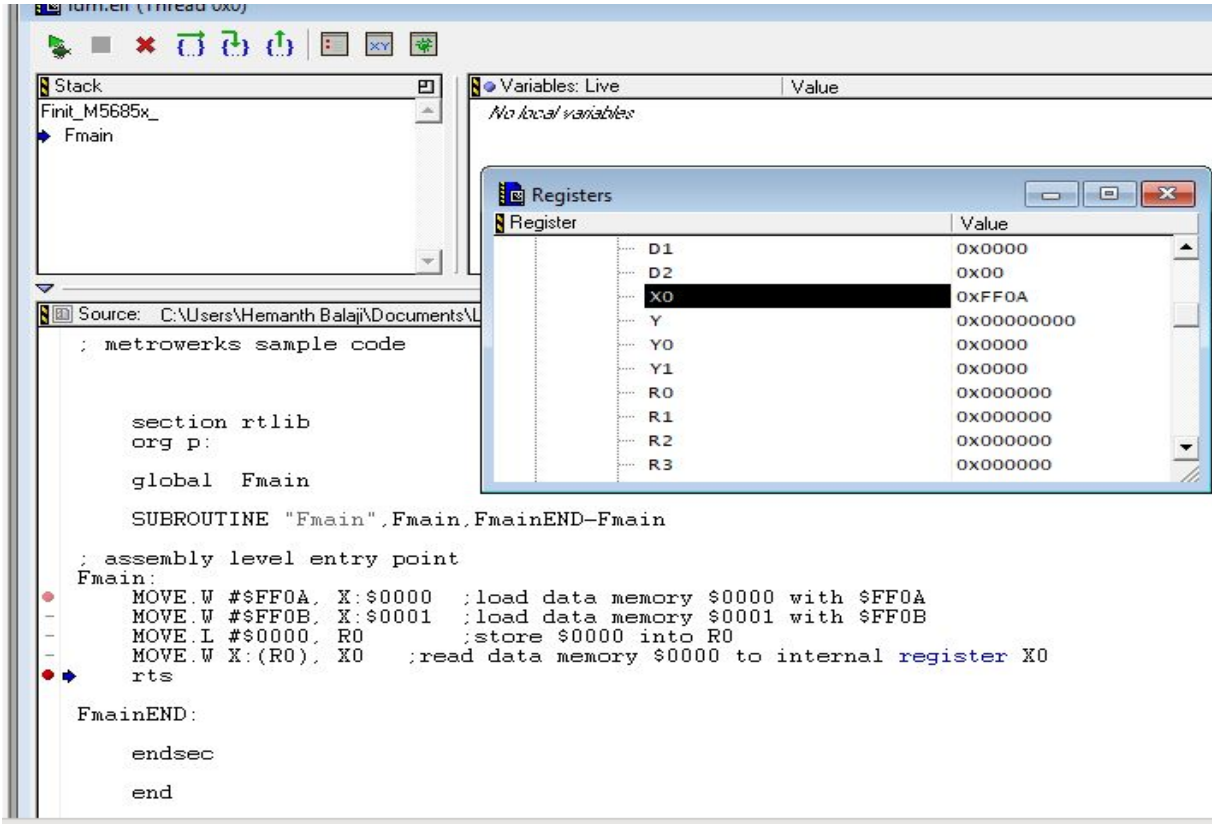


Value of Accumulator A



Value of Register R1

Example 5



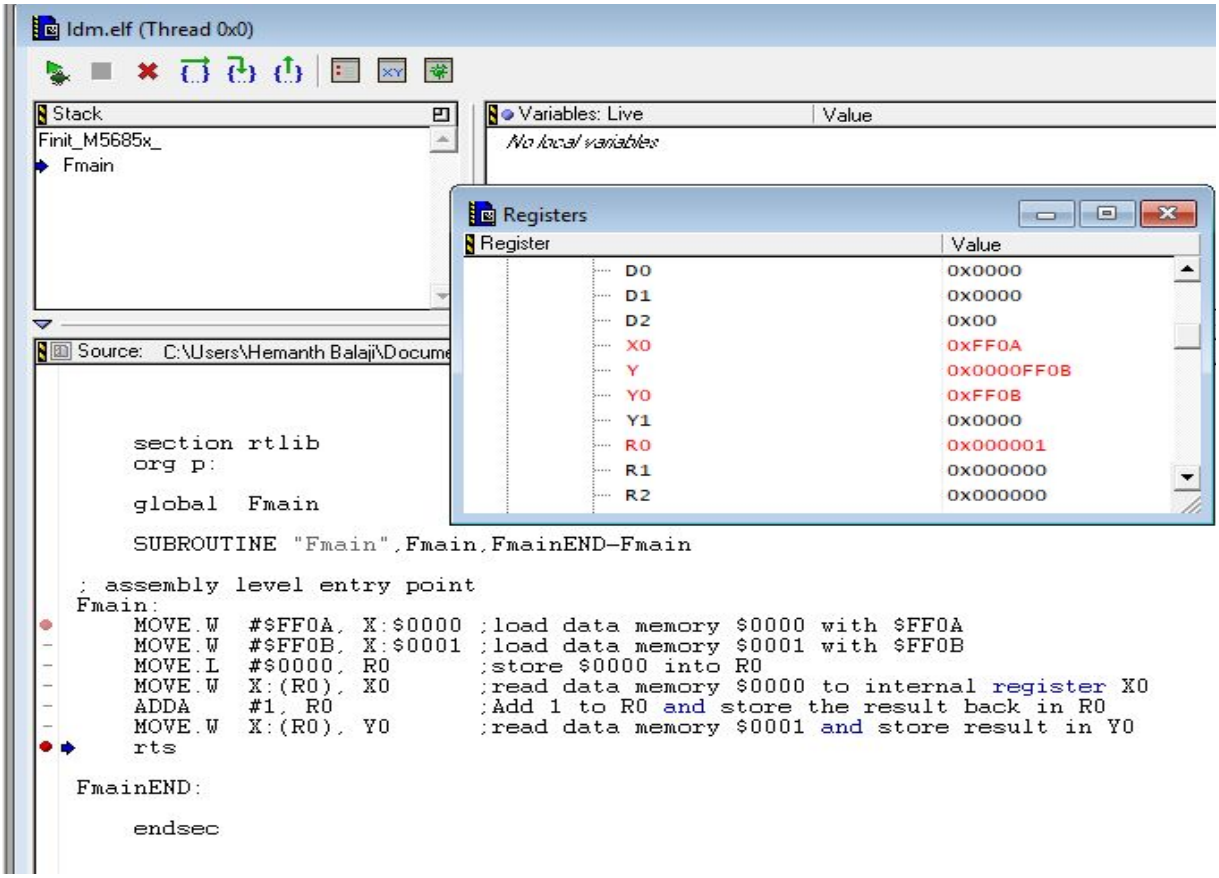
What will be the content of register X0?

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

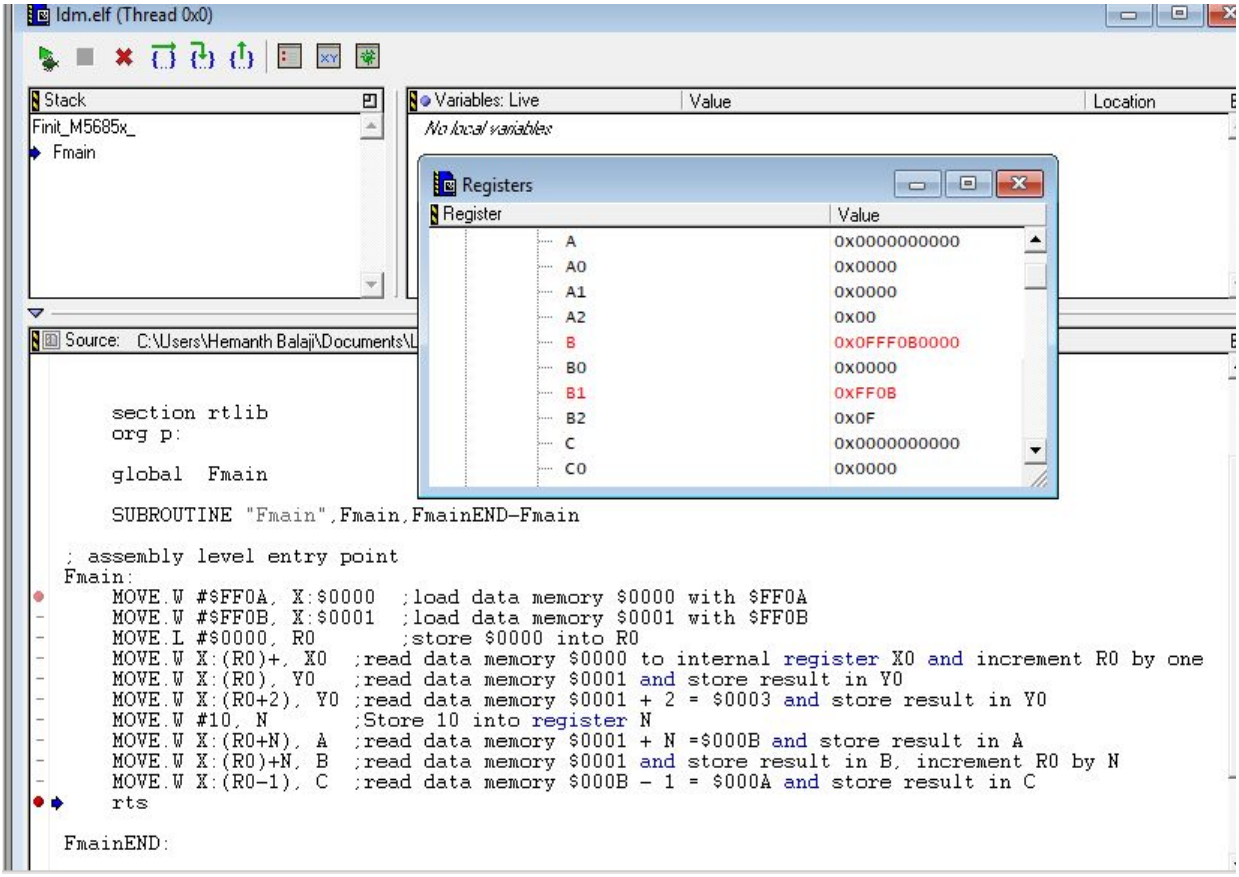
Addressing Modes Used :

- ***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)



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



The cycles and instruction counts are reduced because

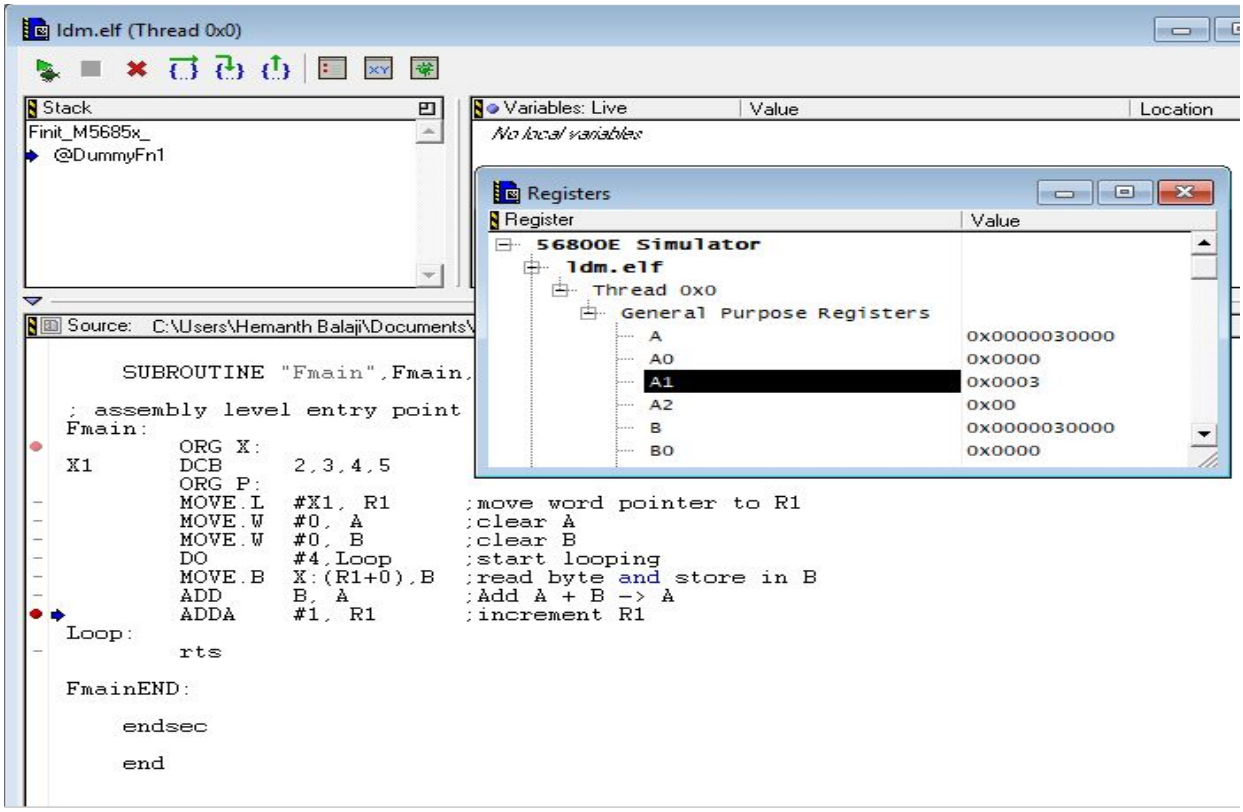
- We replace the
  MOVE.W X:(R0),B
  ADDA N,R0
  by
  MOVE.W X: (R0)+N, **B**

  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.

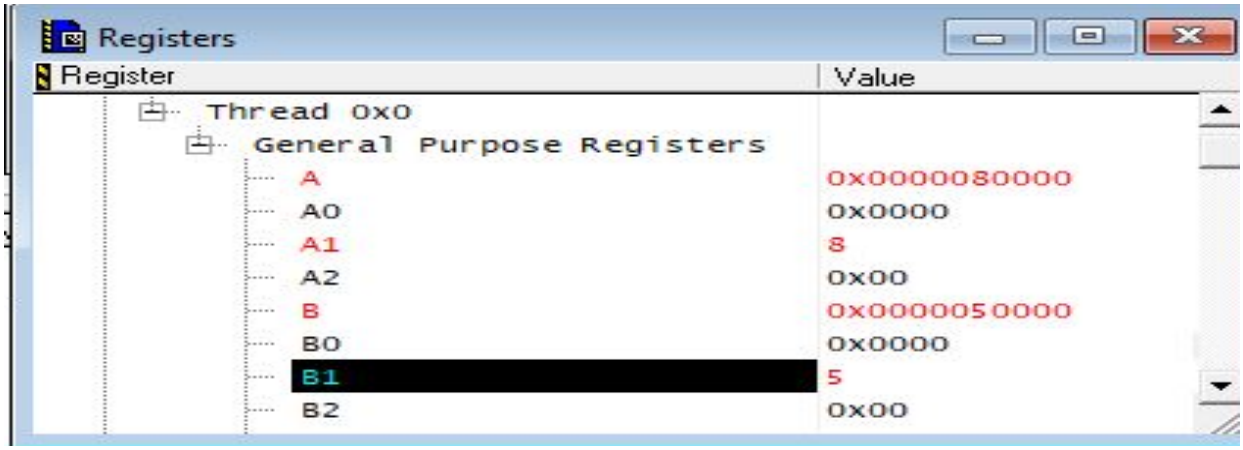
Example 8 (To compare the advantages of MOVE.BP over MOVE.B)

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

Program without using MOVE.BP



First prints 3

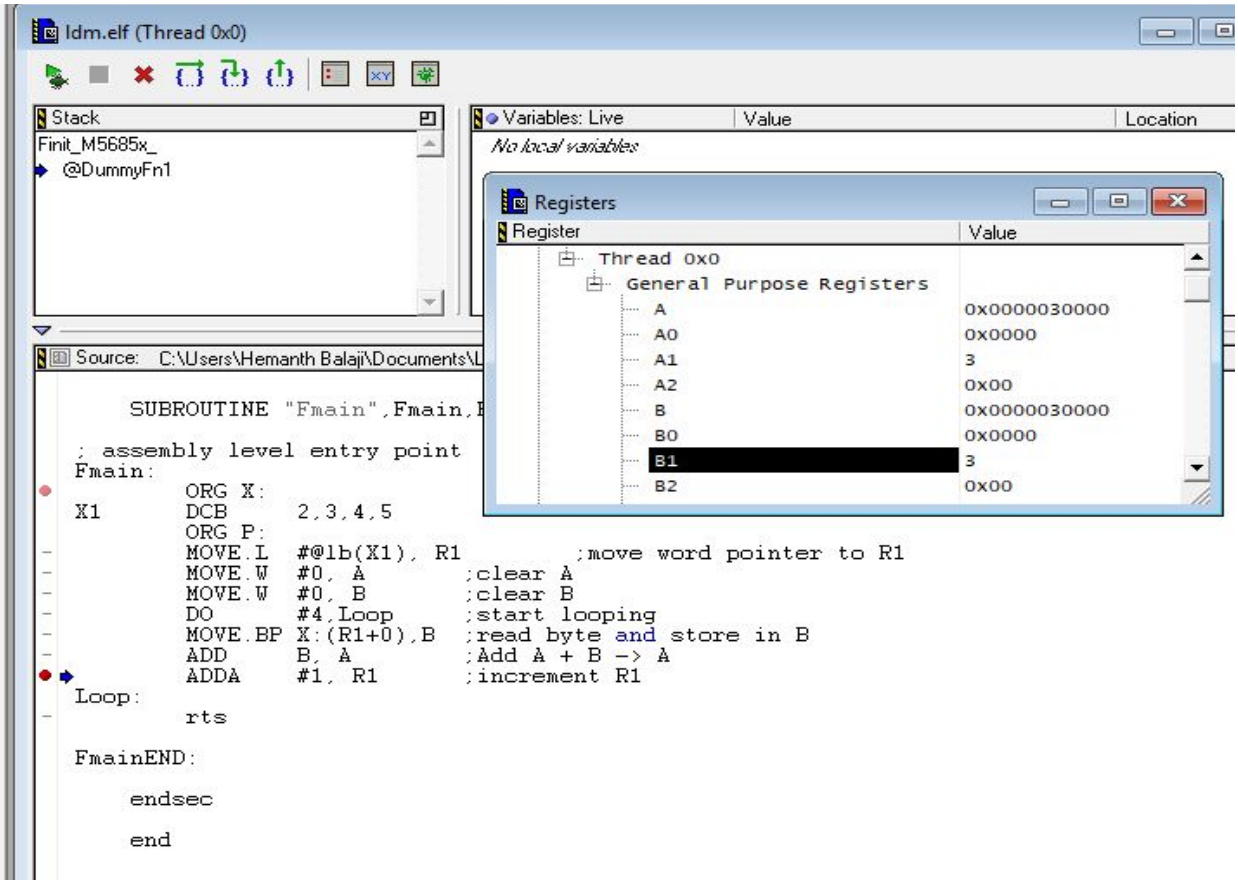


Then skips 2, and Prints 5

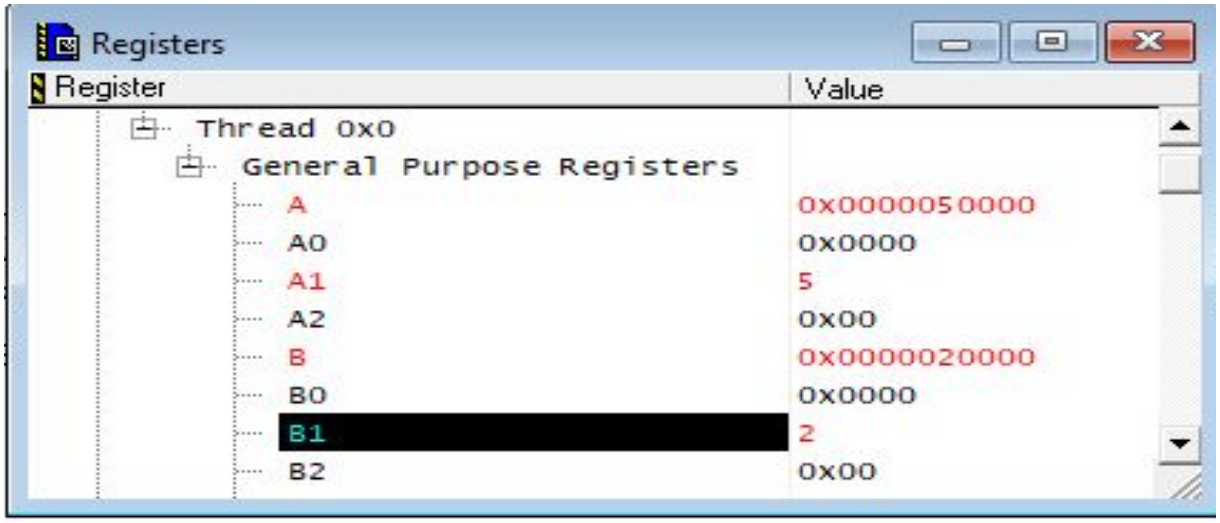
The problem with the above code is that, it just prints out one Byte from the respective words, ie, the lower bytes of each word, thereby skipping the other bytes of each word,

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.

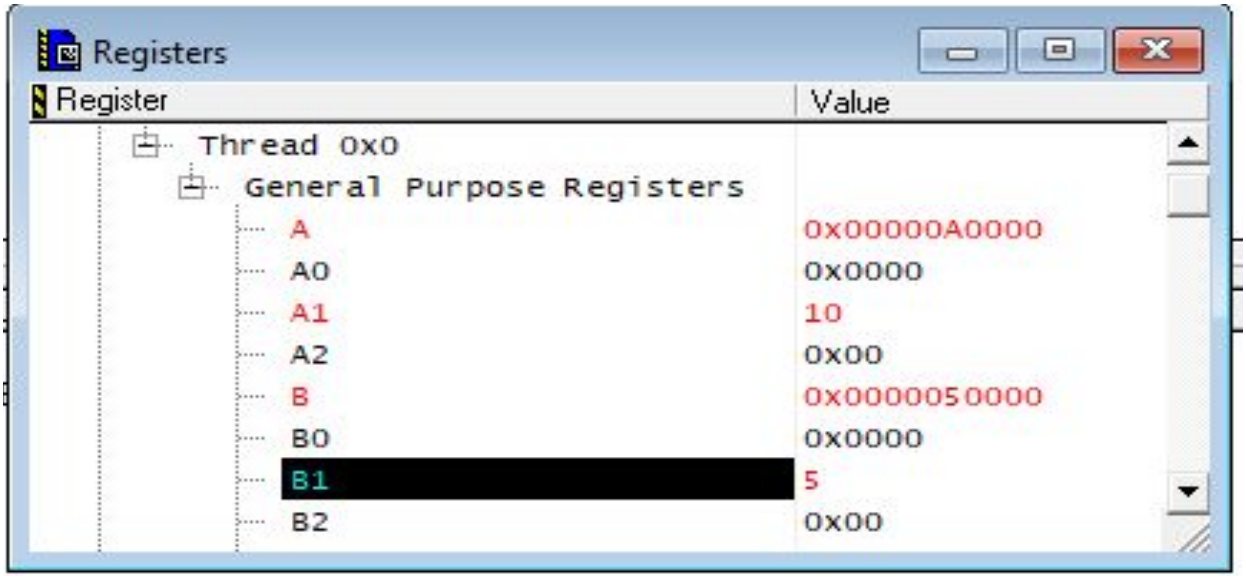
Example 8 (*Program using MOVE.BP*)



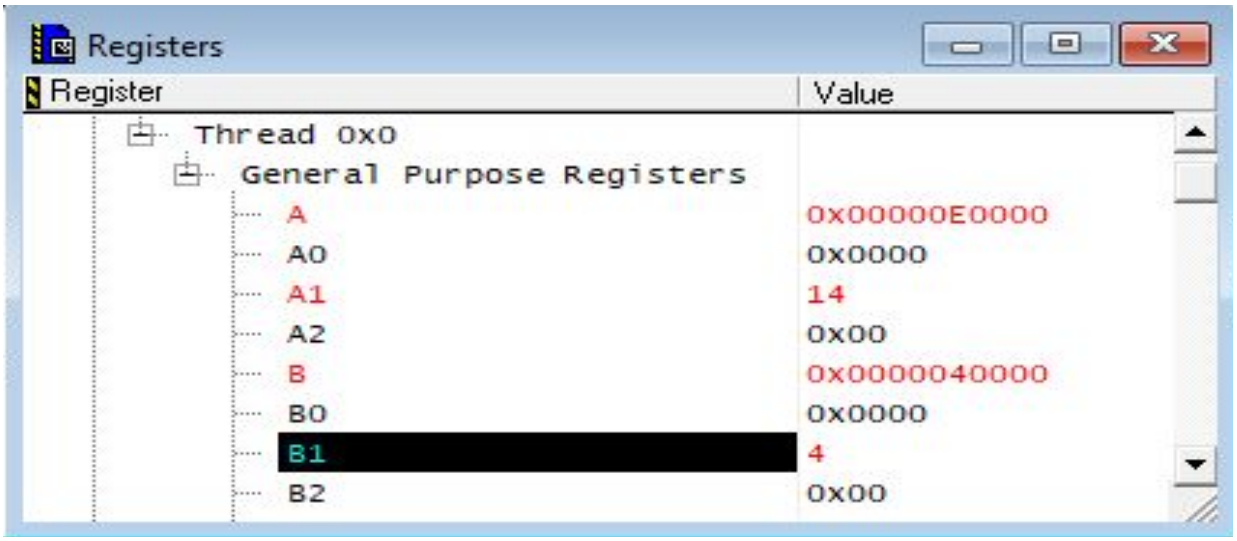
First prints 3



Then prints 2



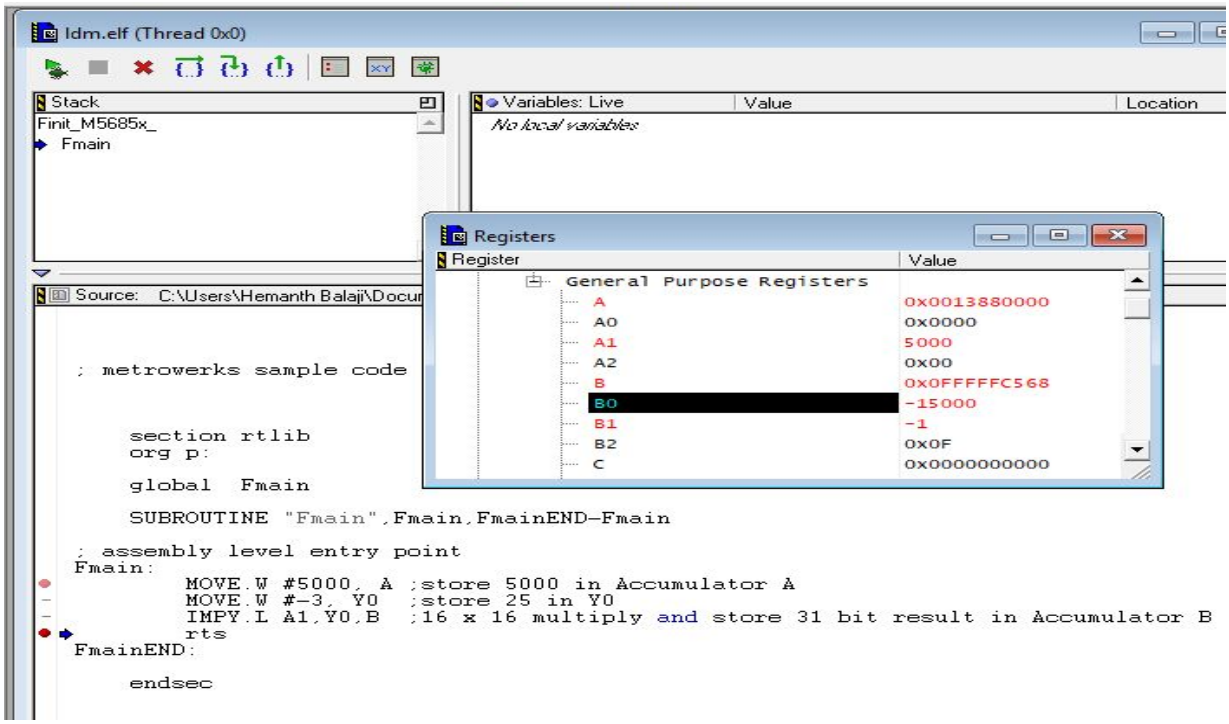
Then Prints 5



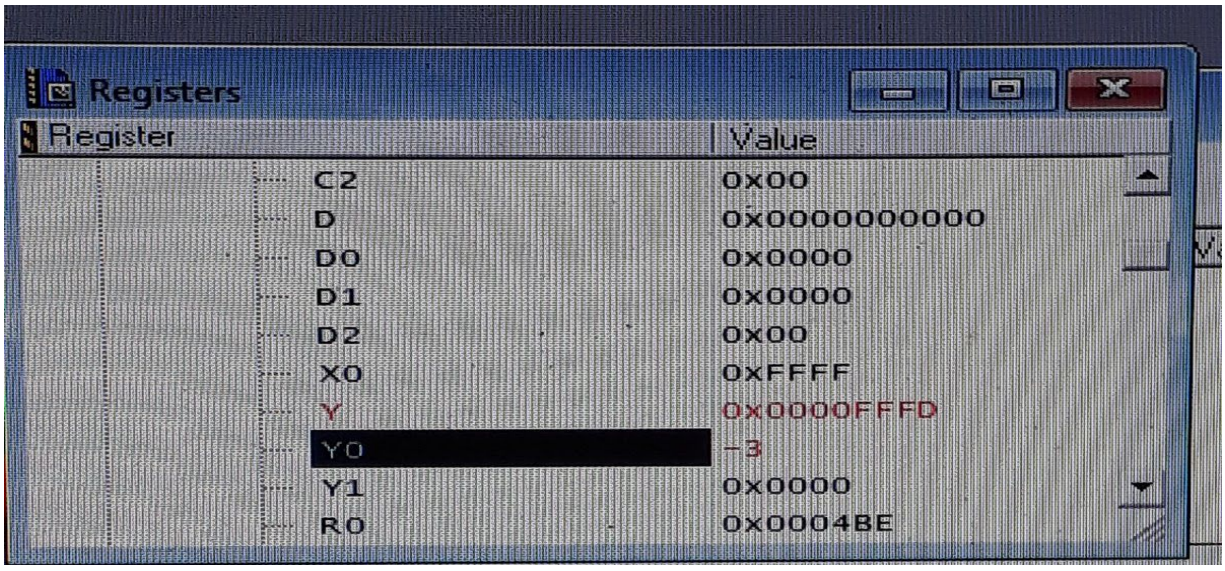
Then Prints 4

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

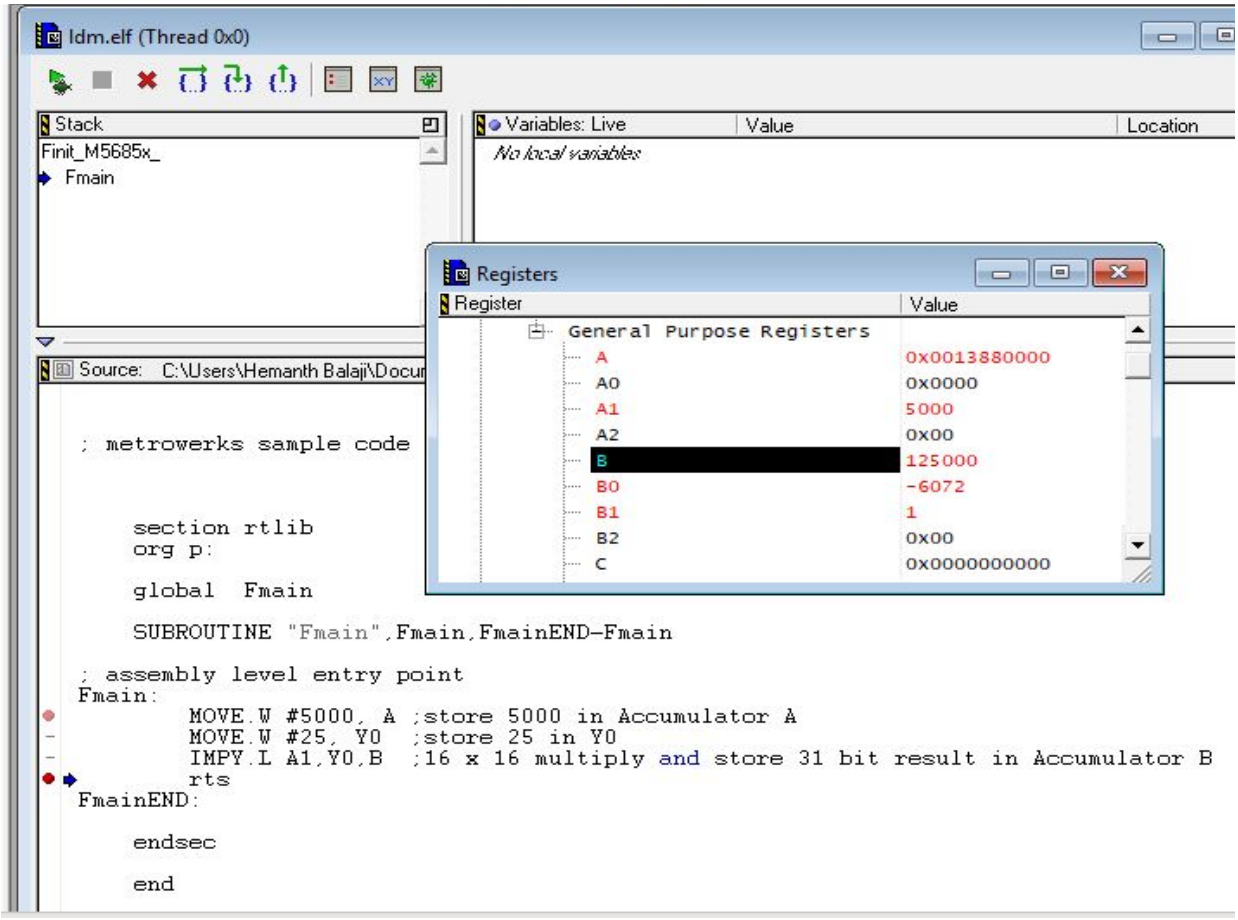


Value stored in Y0

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.***

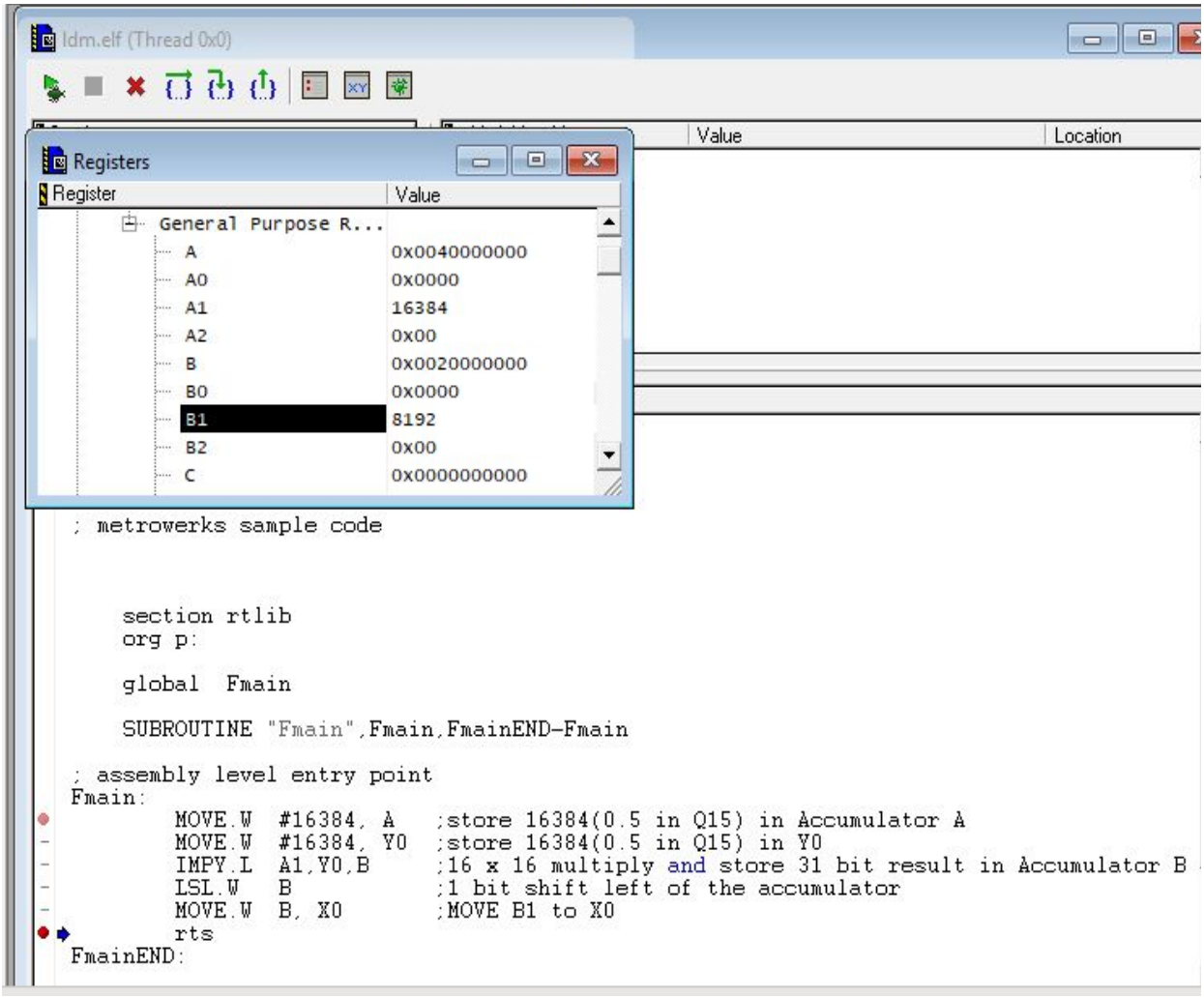
Example 10



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

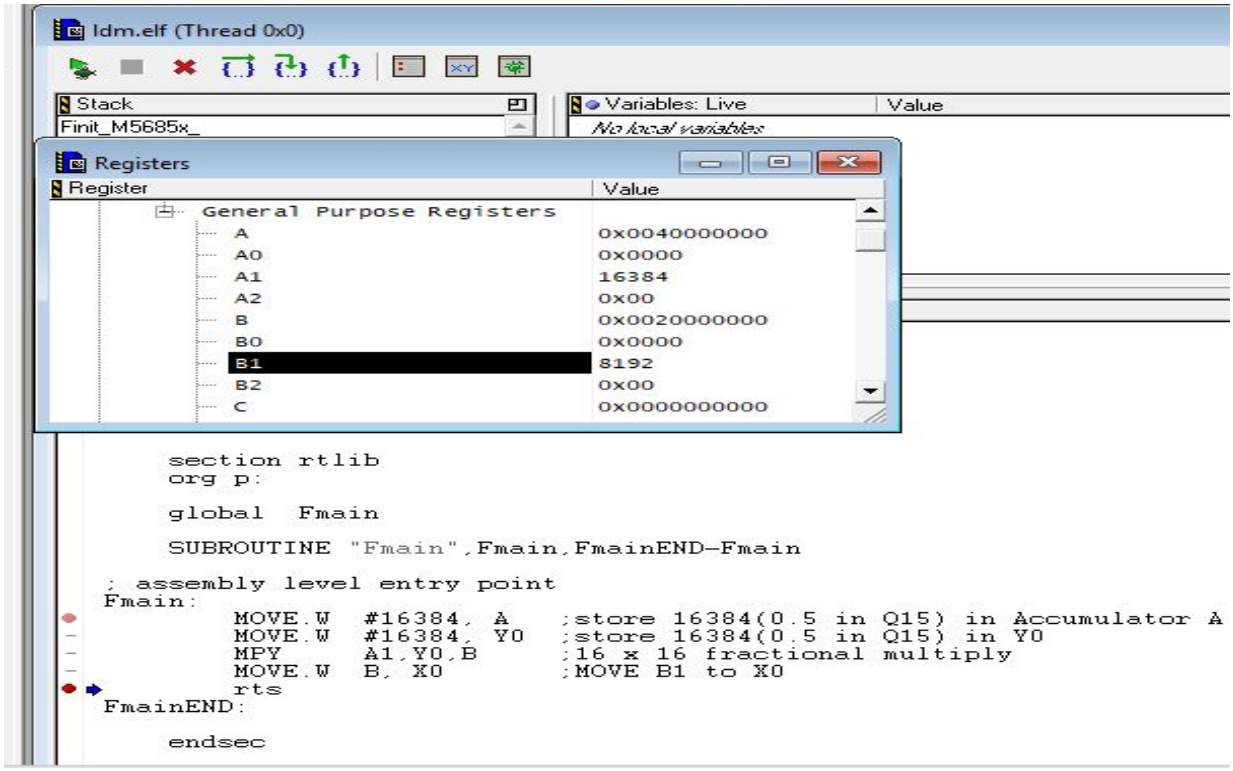
Check the values in B1, B0 and X0. What is the result of the multiplication?

- ***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.***

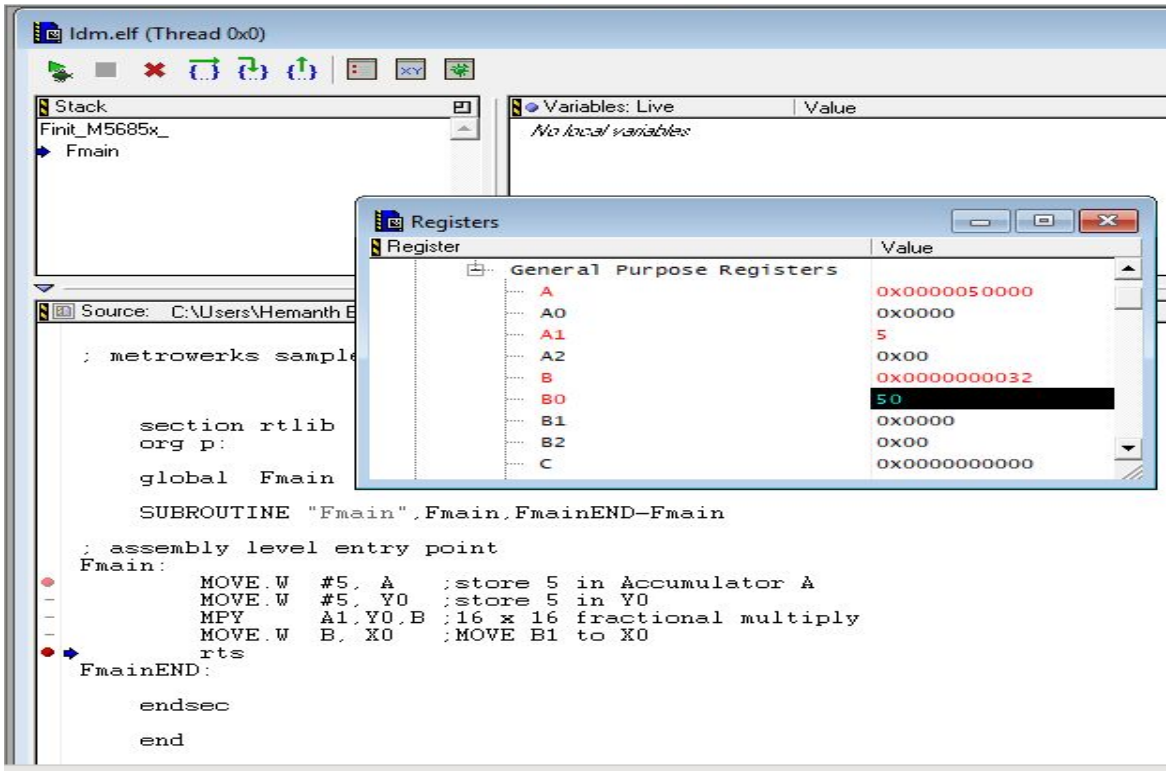
Example 12-To save cycles, we use MPY for Fractional Multiplication



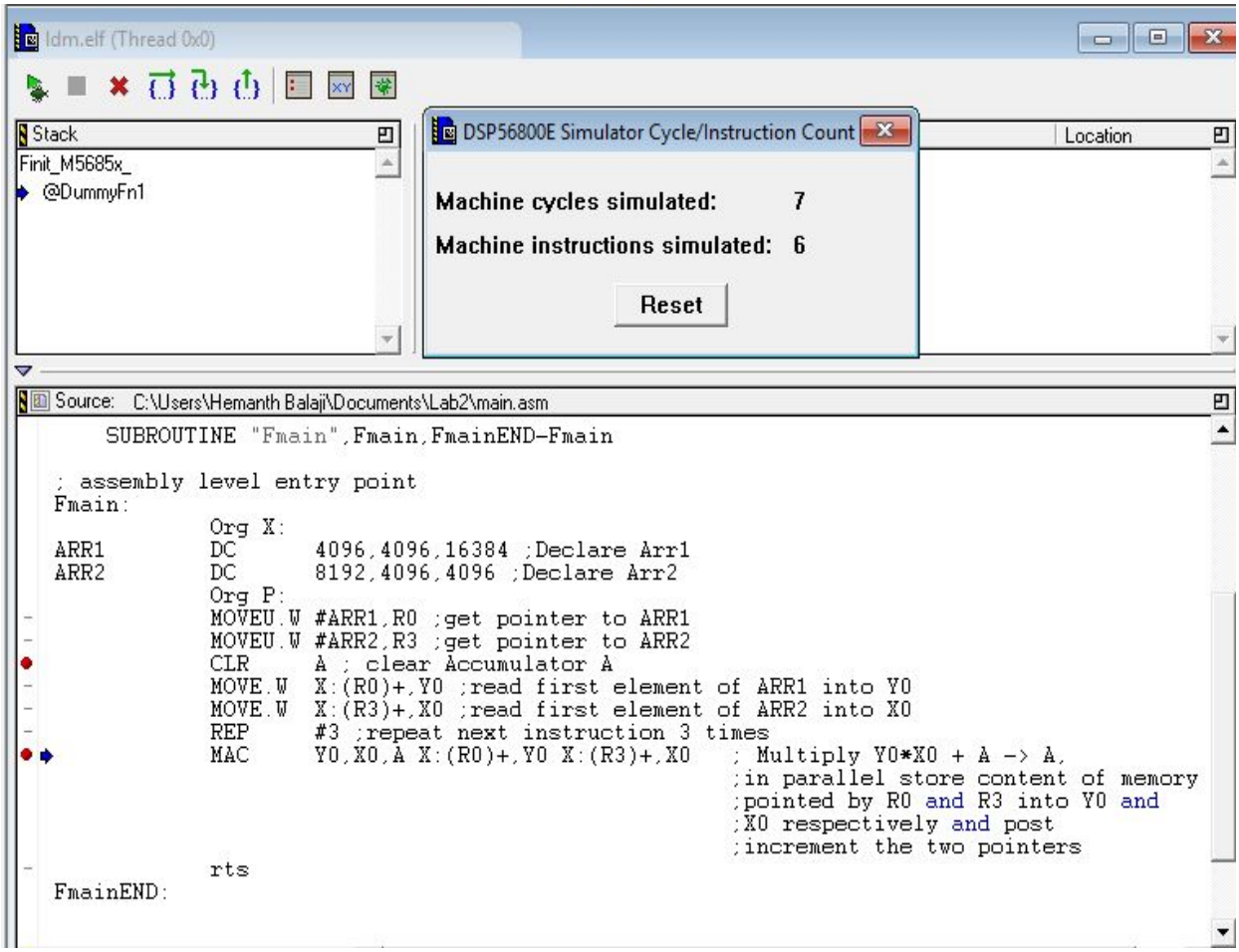
Check again for the result in B1, B0 and X0. Is it the same as the one previously obtained?

- ***Yes, the values are exactly the same.***

Fractional Multiplication b/w 5/2^15 and 5/2^15



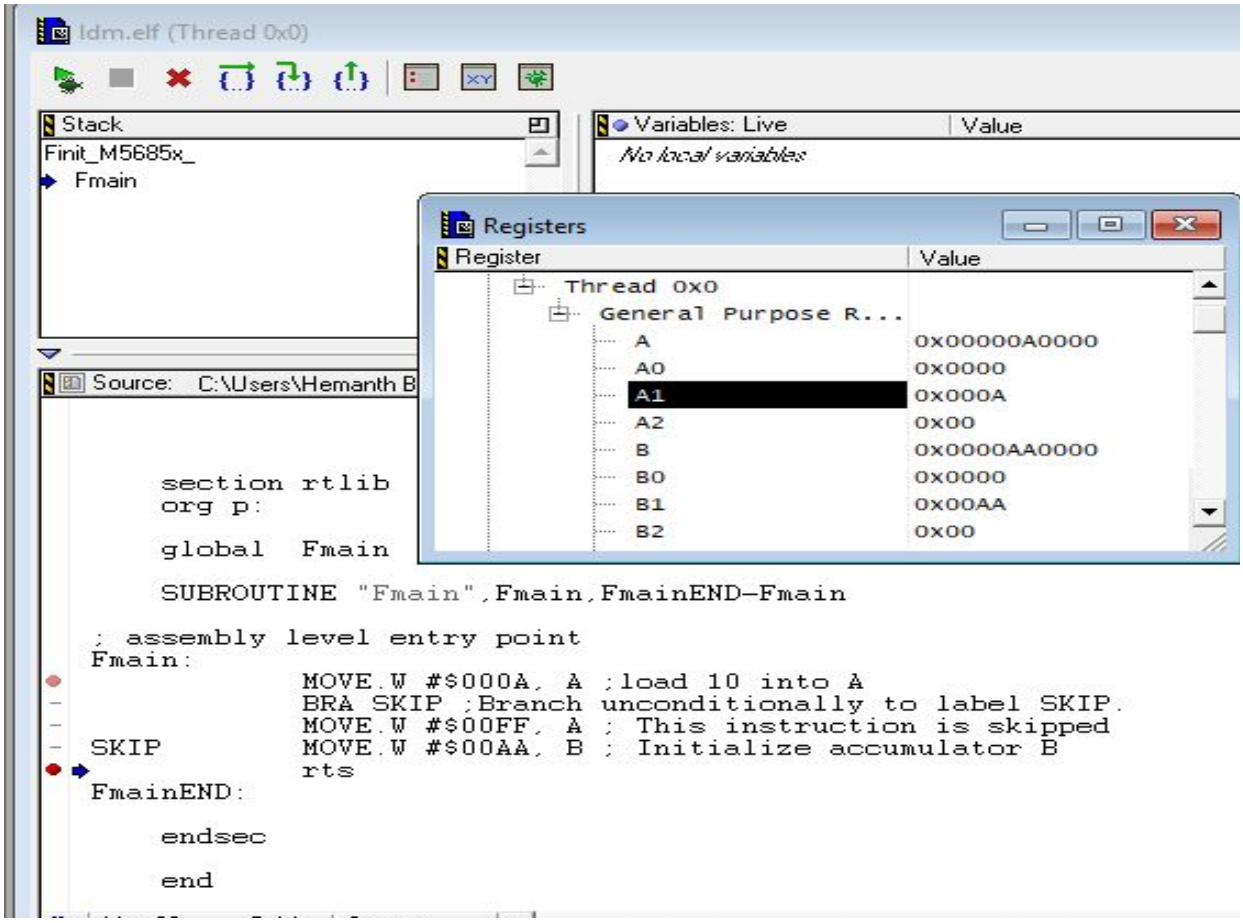
Example 13 (MAC Operation- Multiply and Add Operation)



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*3= 591**	**7**
Machine Instruction count	**101*3=303**	**6**

Example 14

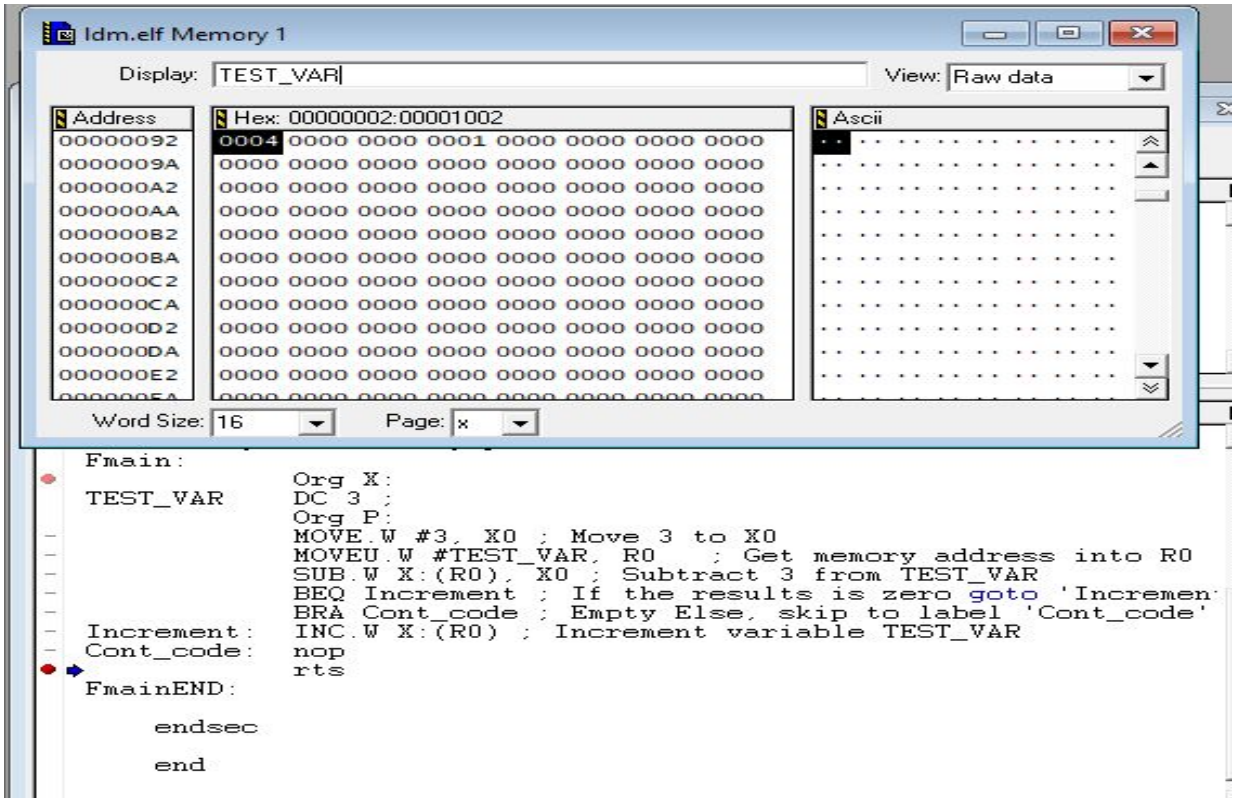


Unconditional branching

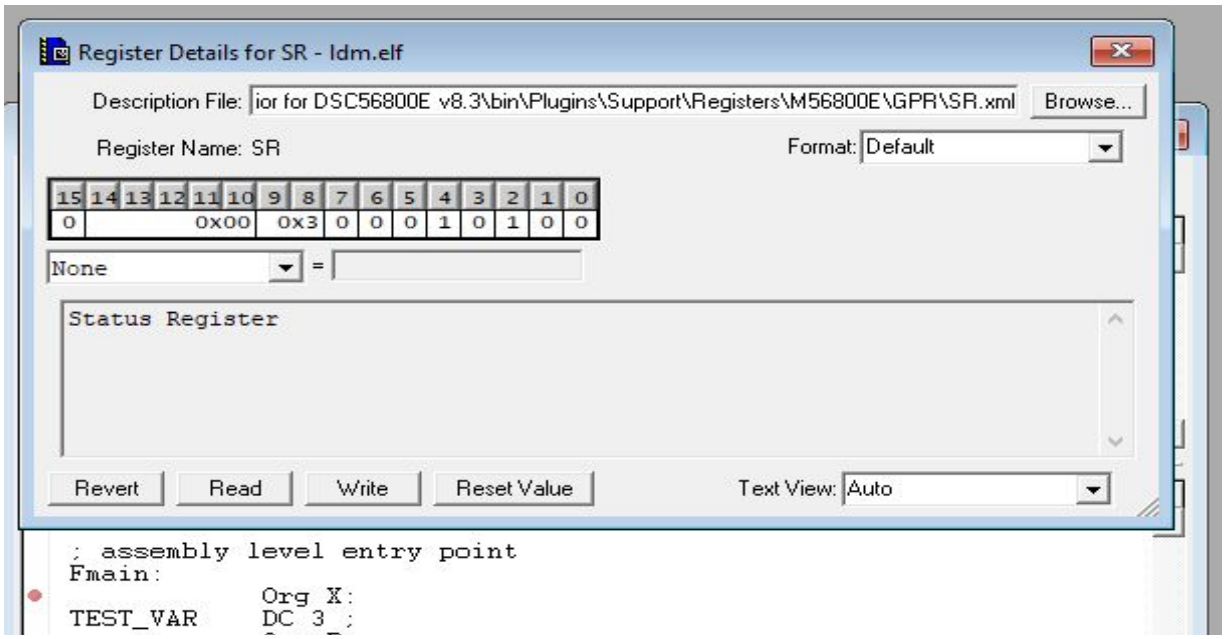
What is the content of accumulator A after code execution?

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

Example 15 Conditional Branching



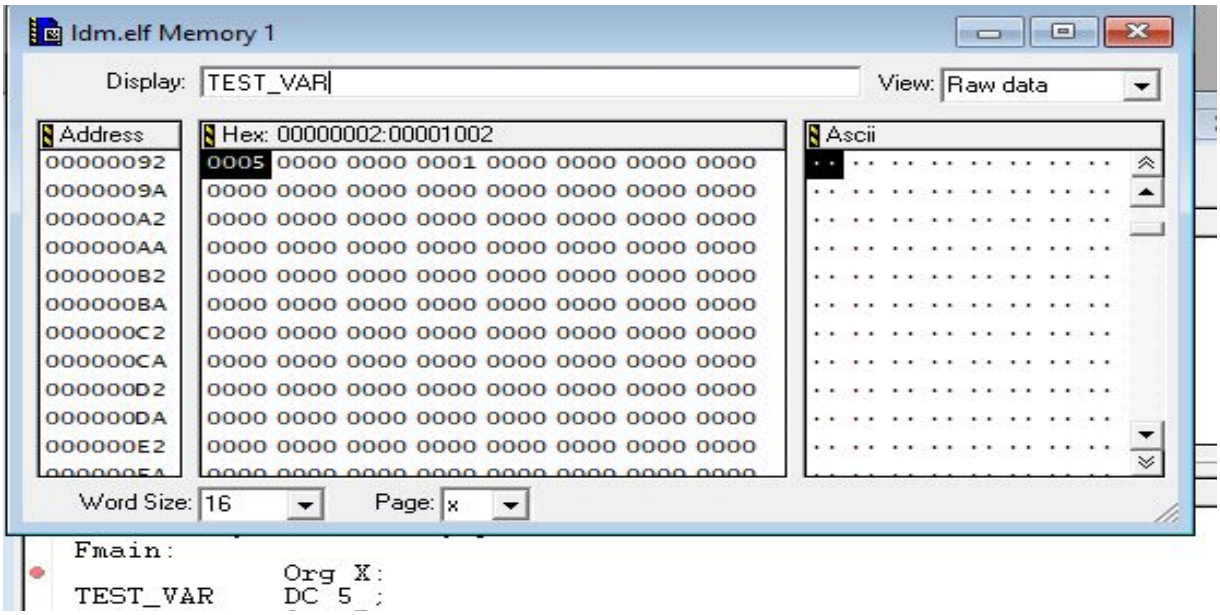
TEST_VAR Final Value



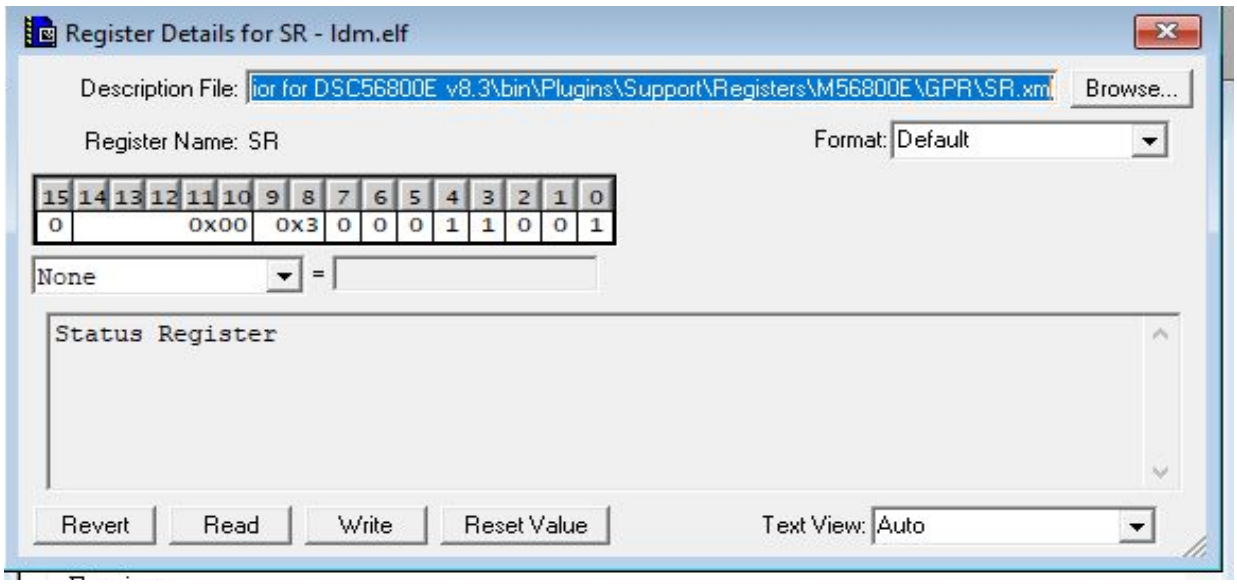
Checking the Status Register for the value of the Z bit.

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

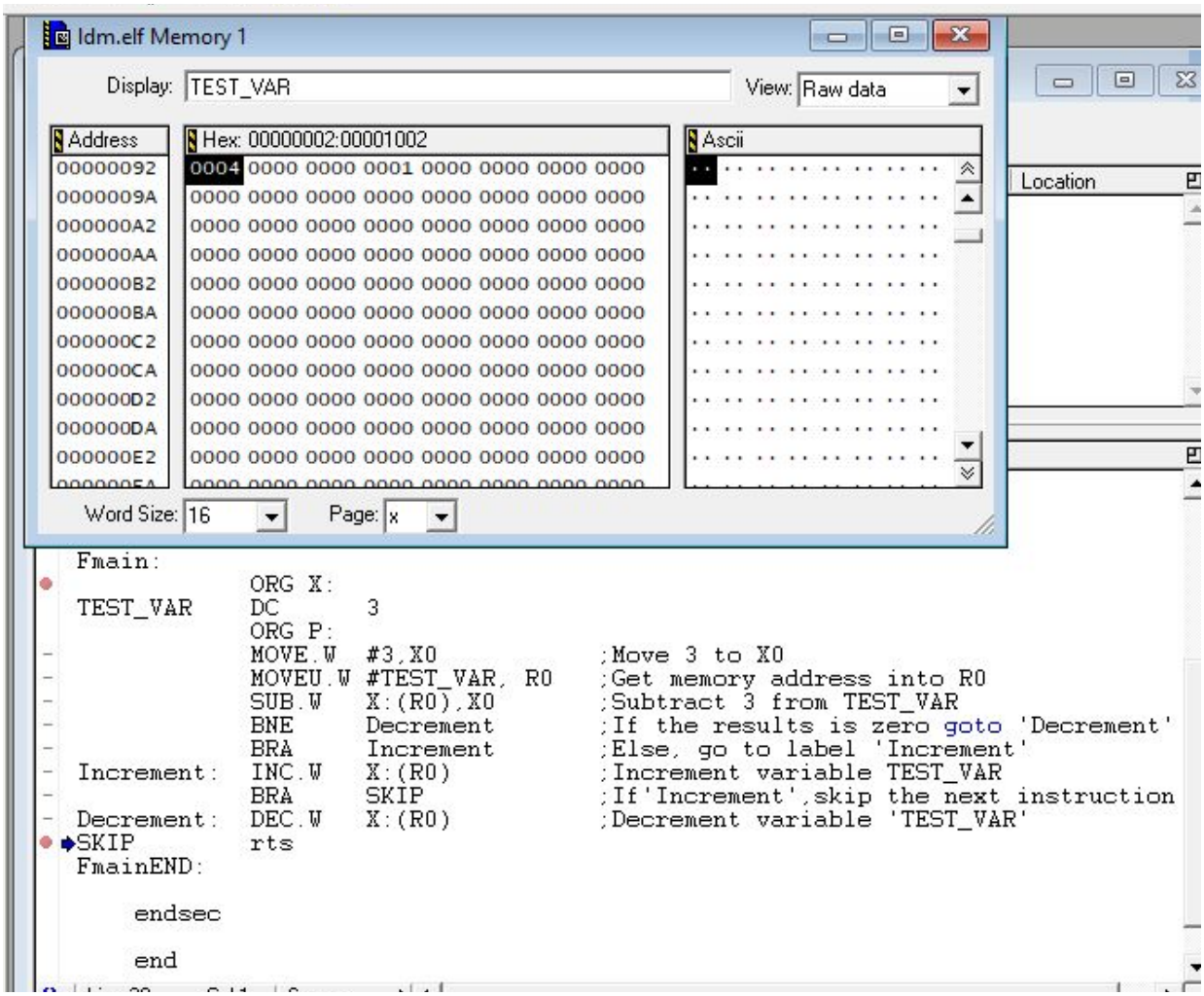
Using BNE *(Grad Only)*

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

x++;

else

x--;}



Example 17

```
    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
```

DSP56800E Simulator Cycle/Instruction Count

Machine cycles simulated: 136

Machine instructions simulated: 60

Reset

Machine Cycles and Instruction Count

ldm.elf Memory 1

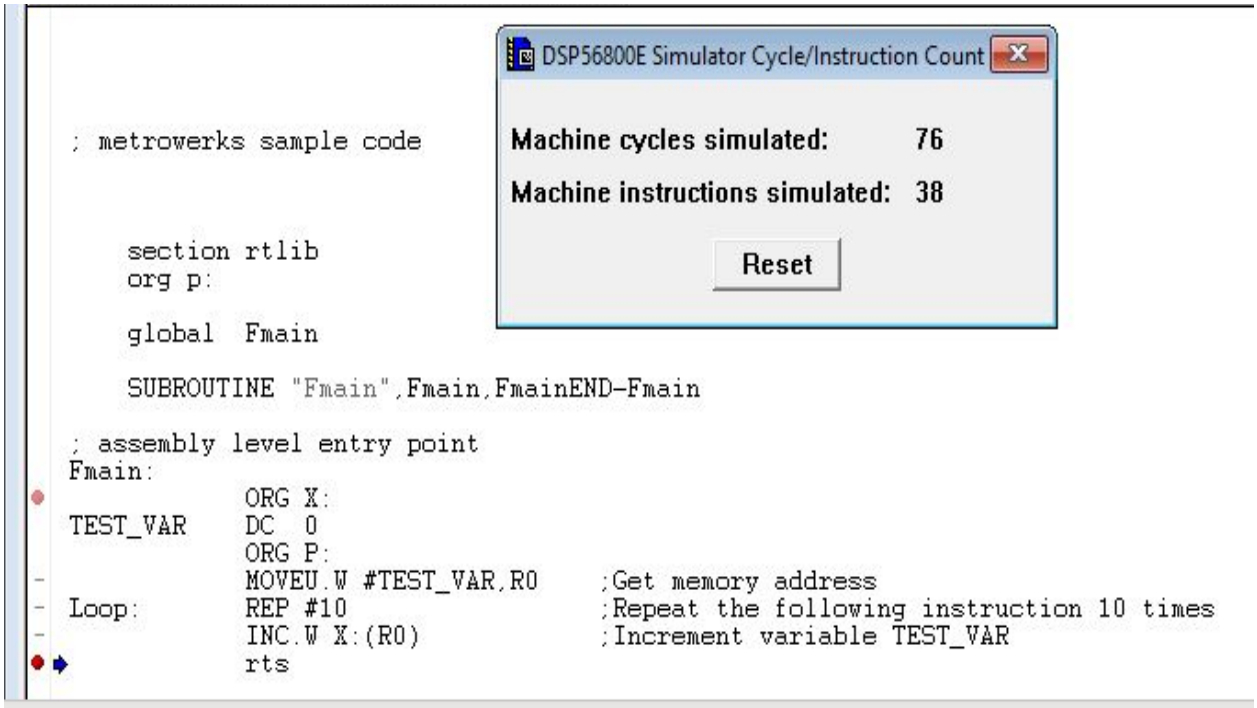
Display: TEST_VAR View: Raw data

Address	Hex: 00000002:00001002	Ascii
00000092	000A 0000 0000 0001 0000 0000 0000 0000	..
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	

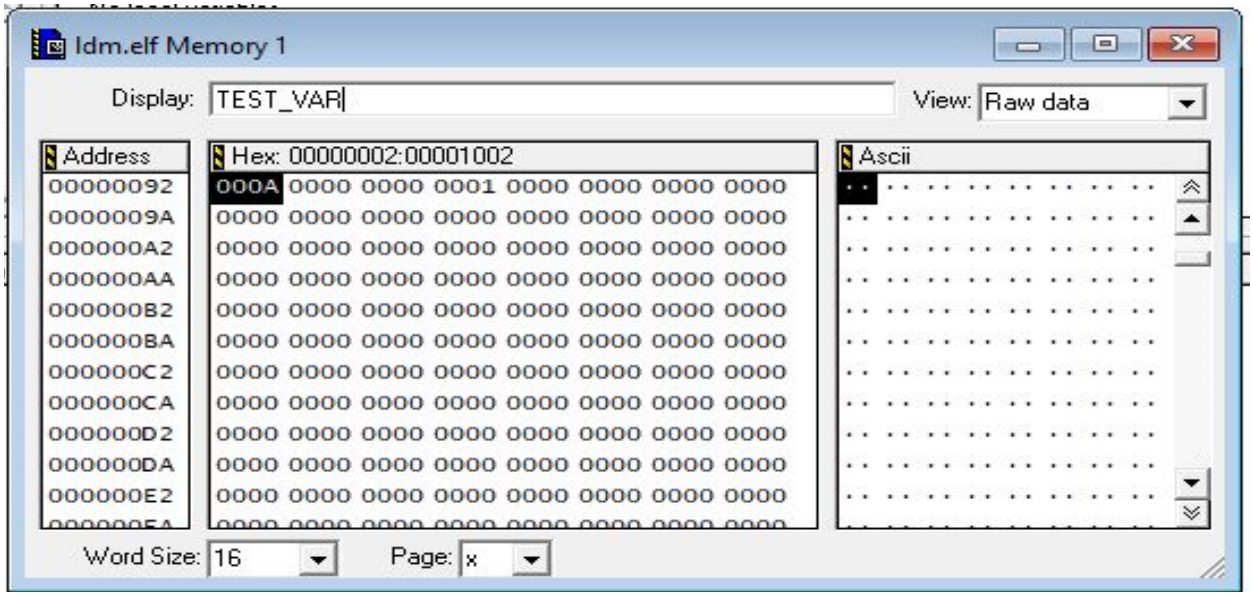
Word Size: 16 Page: x

Value of TEST_VAR

Example 18



Machine Cycles and Instruction Count



Value of TEST_VAR

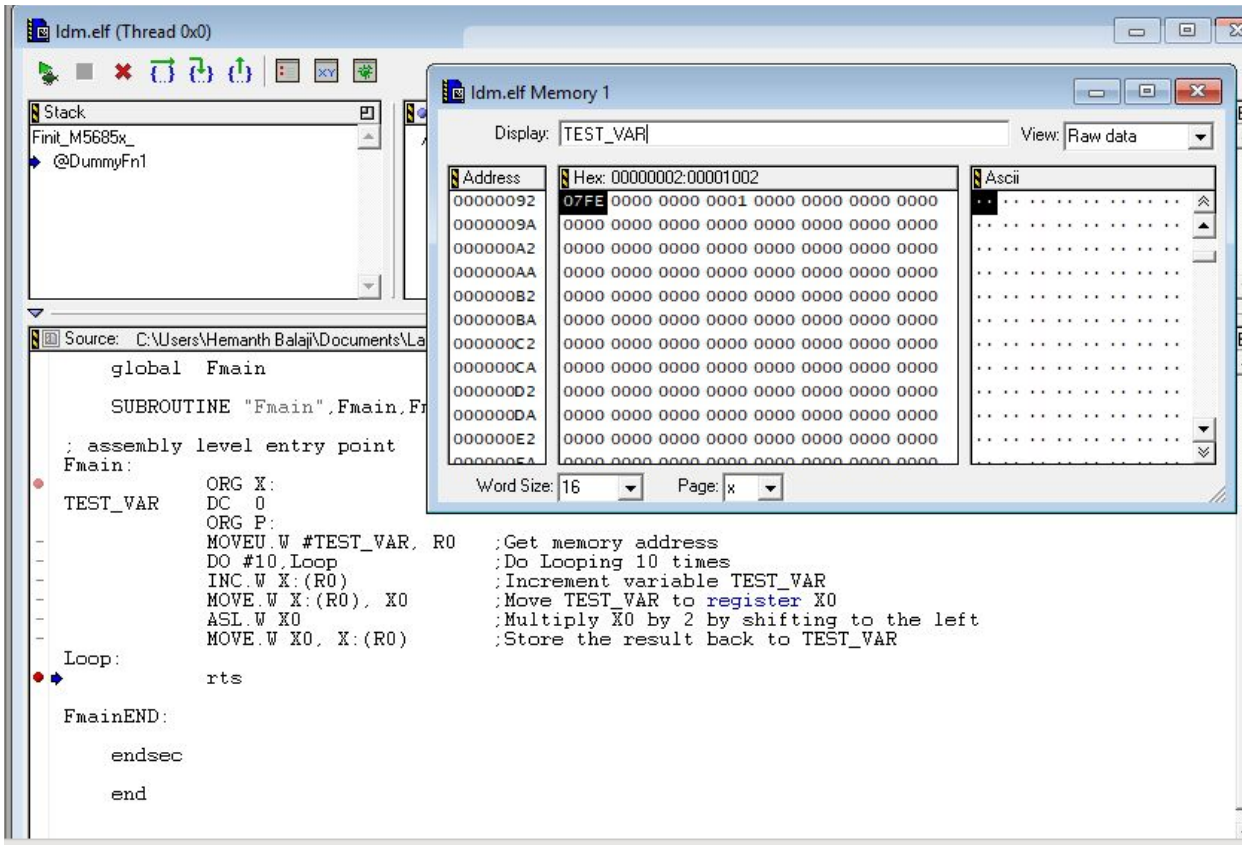
(Grad Only)

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



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!*