Project Report on
ECE 2504 Design Project 3:
16-Bit Arithmetic

Prepared by

[redacted]

CRN [redacted] MWF 09:05

Submitted to

[redacted]

Problem Statement

Intent:
To implement an assembly language subroutine that sums an array of 16-bit two's complement format binary numbers.

Deliverables:

1. Program code:
   1.1. Main Routine should call a subroutine to handle array addition;
   1.2. The subroutine F_Add() shall perform addition on an fixed array (of 16 elements) of 16-bit two's complement numbers using 8-bit arithmetic operations;
   1.3. Main Routine should compute the average value of array elements; and
   1.4. Program should halted upon detection of addition overflow.

2. Listing Files:
   2.1. The listing file should show the final result of program execution;
   2.2. "Add_rv" would contain the sum of the addition, or "0xDEAD" in the case of overflow; and
   2.3. "Average" should contain the average of the sum.

3. Program Verification Plan:
   3.1. A detailed description of the test cases used to verify program functionality.

Design Procedure

The project was broken down into four main objectives, namely, the loop-within code block; the loop-without code block; the average-of-16 code; and the overflow detection subroutine.

The loop-without code block consisted of programming statements that resided in the F_Add subroutine, but outside of the two loops; it also included the preceding statements in the main routine. These statements were responsible for initializing registers and variables to correct values prior to any function call or loops.

The loop-within code block resided in the F_Add subroutine and basically implemented the summation algorithm. It was responsible for the 8-bit arithmetic operations needed to add an array of 16-bit numbers accurately.

The overflow detection subroutine OF_Detect was called by F_Add after every summation; this was needed to check the accuracy of the addition. Should there be an overflow, there would be no further need to continue with the summation process. Upon receiving the "break" signal from OF_Detect, F_Add would return with the appropriate error flag, and consequently, the main routine would terminate.

The Average-of-16 code block was responsible for calculating the average of the sum. Using a combination of nibble-swapping, bit-masking and various bit-wise operations (OR, AND, XOR), the code block executed the equivalent of four consecutive arithmetic right-shifts.

Flowcharts

Flowchart for main() routine

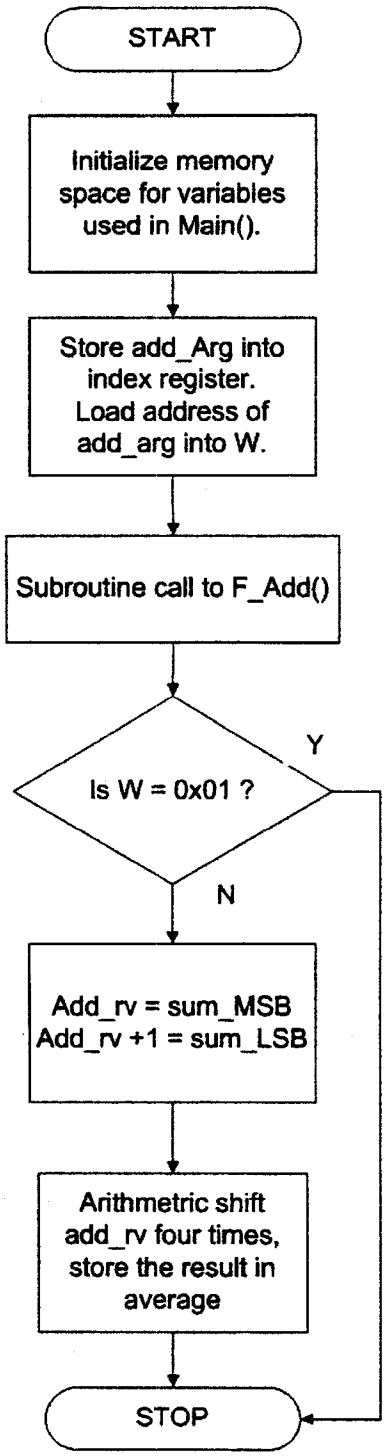


Fig. 1: Flowchart for the main routine

Flowchart for F_Add() subroutine

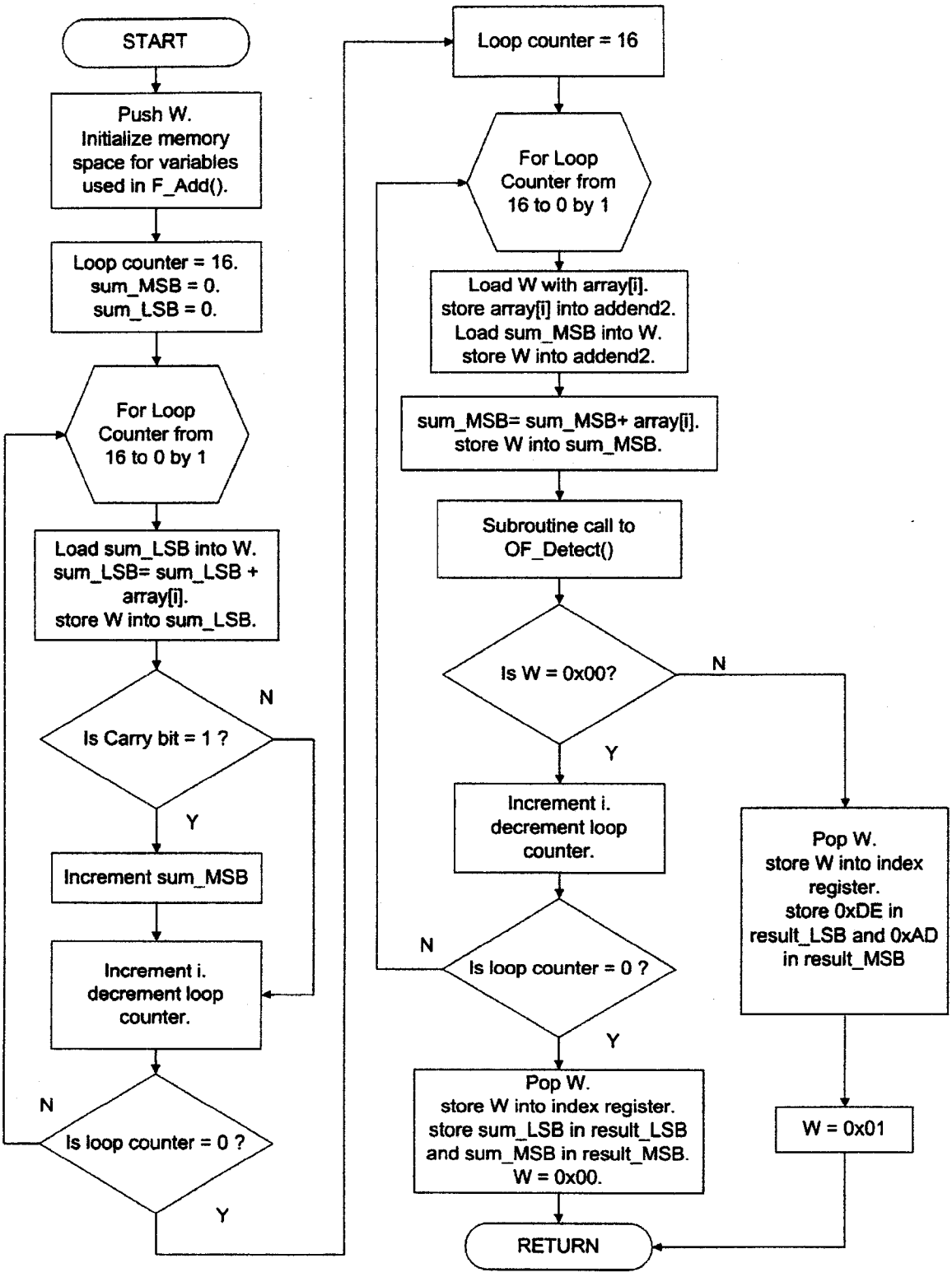


Fig. 2: Flowchart for the F_Add subroutine

Flowchart for OF_detect() subroutine

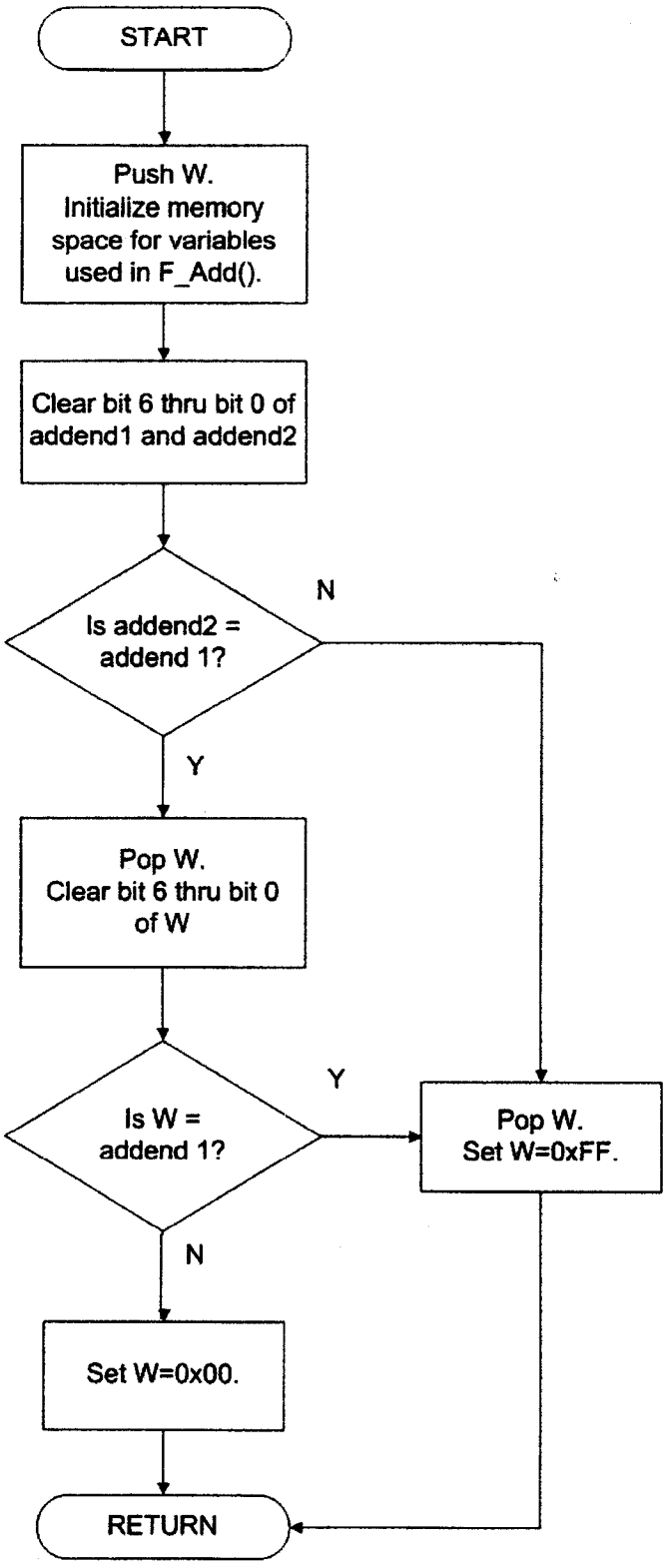


Fig. 3: Flowchart for the OF_Detect subroutine

Implementation

Basis of using subroutines

Sub-routines are frequently used to divide-and-conquer a programming problem, and this project was no exception. The only additional subroutine defined was the overflow detection routine OF_Detect. Because overflow has to be checked after every addition of the upper, more significant bytes, OF_Detect is called repeatedly, once during each loop-run. In order to simplify coding, as well as to shorten code length, an additional overflow detection subroutine was coded.

File register locations

Table 1: Addresses of used memory locations

Location Label	Location Address	Remarks
Add_arg	0x10	Contains the array. Occupies 32 memory locations from 0x10 through 0x2F inclusive.
Add_rv	0x30	Contains the result of array summation. Occupies 2 memory locations 0x30 and 0x31.
average	0x32	Contains the average from array summation. Occupies 2 memory locations 0x32 and 0x33.
sum	0x40	Contains the result of array summation for use in F_Add subroutine.. Occupies 2 memory locations 0x40 and 0x41
Add_Result	0x42	Contains the address of Add_rv. It held a value of 0x30 constantly through this project.
loopCounter	0x43	Contains the loop counter for use in program loops.
Addend1	0x44	Contains the MSB of an array element. For use in overflow detection
Addend2	0x45	Contains the MSB of the sum after every addition. For use in overflow detection, if required.
tempByte	0x46	A space to hold temp 8-bit values. It is sometimes used as a "stack" to preserve the value of the working register in subroutines

Program Verification Plan

As the highest value in 16-bit 2's complement is 0x7FFF and the lowest, 0x8000, "nibbles of interest" were determined to be limited to 0x00, 0x7F and 0x80.

For ease of testing, the first test number (Addend1) was placed at the first location of the array and the second number (Addend2) was placed at the last location of the array. As the numbers in the other locations were set to zero, this ensured that the program goes through all sixteen loops in the F_Add subroutine.

Proper loop iteration was also verified by observing data location 0x43 (variable "loopCounter"); it was 0x00 when no overflow was detected (inferring that the loop counter had been decremented to zero) and 0x01 when overflow was detected (thus inferring that an overflow break had occurred in the middle of its last loop).

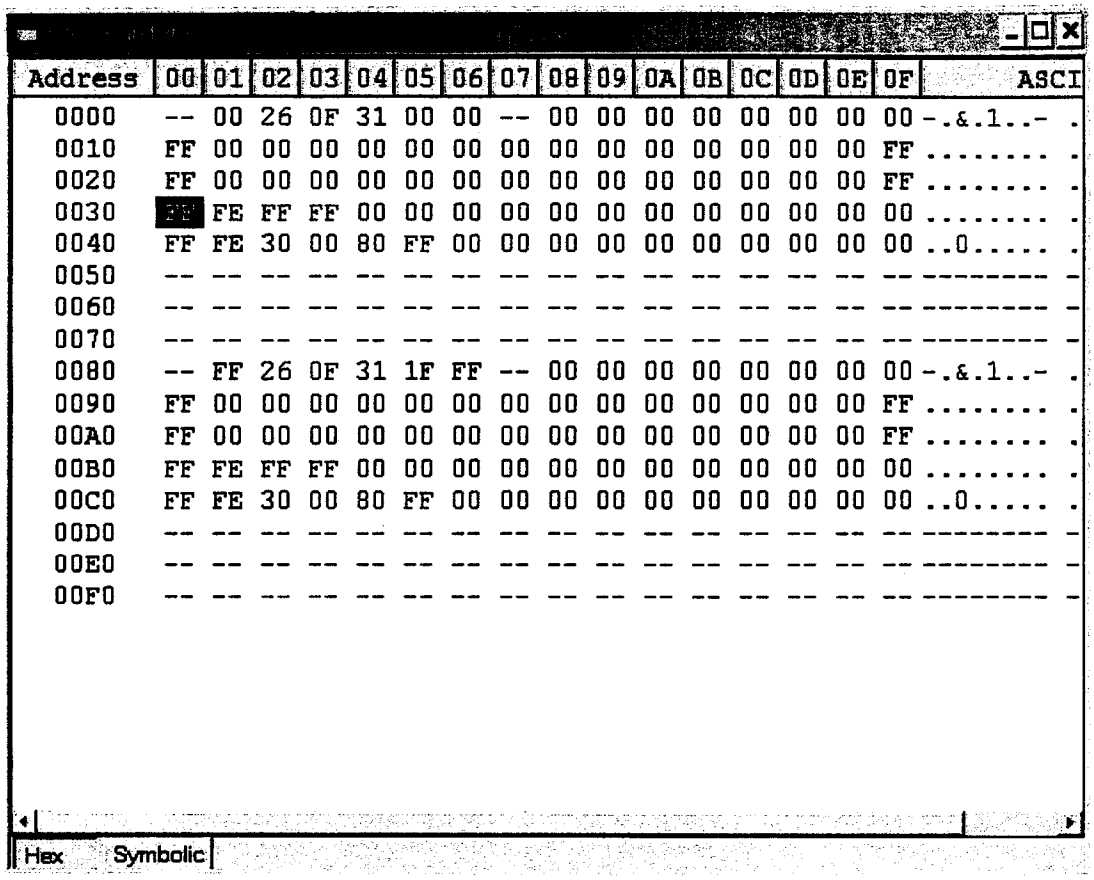
Test Case #5 was an exception. In this test case, addend2 was stored in the second array location. This not only served as a reconfirmation of proper array addition (but observing locations 0x40 and 0x41) and averaging, but also determined if the program was able to prematurely exit upon overflow detection.

Table 2: Test Cases

Test Case	Addend1	Addend2	Add_arg contents	Add_rv contents	Remarks
1	0xFFFF	0xFFFF	0xFFFE	0xFFFF	Verified the proper addition of negative numbers Also verified proper ÷16 operation
2	0x7FFF	0x8000	0xFFFF	0xFFFF	Verified ability to determine no overflow when MSb of both addends were different
3	0x8000	0x8000	0xDEAD	0x0000	Verified ability to determine overflow when MSb of both addends were identical (for negative numbers) Also verified that "average" reported 0x00 when overflow had occurred.
4	0x0000	0x0000	0x0000	0x0000	Verified ability to determine overflow when MSb of both addends were identical (for positive numbers) Also ensured integrity of output variables (inadvertent writing had not occurred)
5	0x70FF	0x70FF	0xDEAD	0x0000	Verified ability to exit program prematurely upon detection of overflow

Test Results

The following is a screen shot of the File Registers window, depicting final result of the program executing test case #1:



Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCI
0000	--	00	26	0F	31	00	00	--	00	00	00	00	00	00	00	00	-.&.1..- .
0010	FF	00	00	00	00	00	00	00	00	00	00	00	00	00	00	FF	..... .
0020	FF	00	00	00	00	00	00	00	00	00	00	00	00	00	00	FF	..... .
0030	FF	FE	FF	FF	00	00	00	00	00	00	00	00	00	00	00	00	..... .
0040	FF	FE	30	00	80	FF	00	00	00	00	00	00	00	00	00	00	..0..... .
0050	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	----- -
0060	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	----- -
0070	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	----- -
0080	--	FF	26	0F	31	1F	FF	--	00	00	00	00	00	00	00	00	-.&.1..- .
0090	FF	00	00	00	00	00	00	00	00	00	00	00	00	00	00	FF	..... .
00A0	FF	00	00	00	00	00	00	00	00	00	00	00	00	00	00	FF	..... .
00B0	FF	FE	FF	FF	00	00	00	00	00	00	00	00	00	00	00	00	..... .
00C0	FF	FE	30	00	80	FF	00	00	00	00	00	00	00	00	00	00	..0..... .
00D0	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	----- -
00E0	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	----- -
00F0	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	----- -

Hex Symbolic

Fig. 4: Screenshot of the File Registers Window

Shown below this a snapshot of the Watch window taken after the execution of test case #1; the working register and other critical memory locations were monitored.

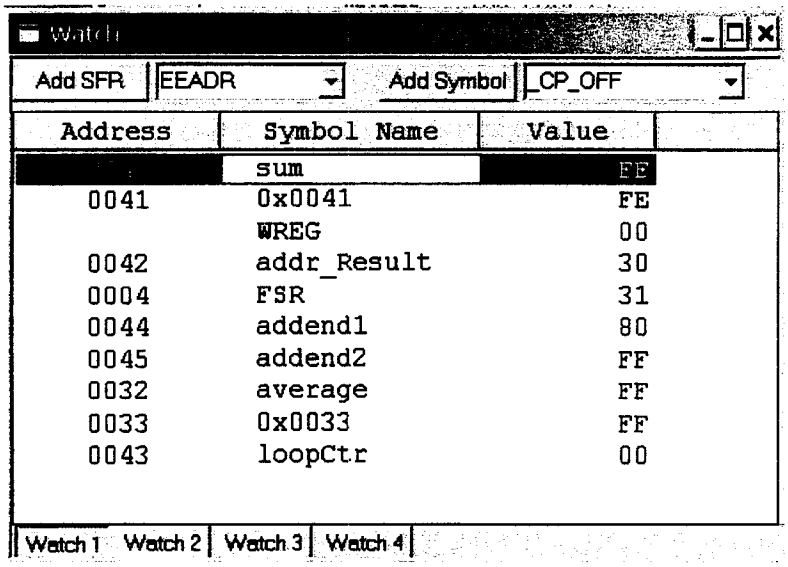


Fig. 5: Screenshot of the Watch Window

Conclusion

The project was considerably more challenging than previous projects due to the introduction of low-level software programming. Assembly programming not only demands an intimate knowledge of the hardware involved, but also the analytical astuteness critical when debugging programming code.

This project served as an excellent primer to the common obstacles in programming embedded devices. Manipulating numbers that "wider" than the working register is a common problem that all CPU low-level programmers face. The 8-bit PIC's lack of multiple accumulators and indexing registers made 16-bit arithmetic a tad more difficult; it also illustrated the importance of code streamlining and memory management.

Deciding whether to implement the big endian or little endian convention was also a consideration. The programmer has to constantly keep track of whether the higher significant values are stored in either higher/lower significant memory locations. Failure to maintain such situational awareness can prove to be problematic.

However, I was slightly perturbed with two areas of the project. Firstly, the overflow detection algorithm as mentioned in the project specifications was incorrect. As it turned out, the carry bit was irrelevant when detecting overflows, rather, the signed bit proved more crucial; Secondly, the specified memory layout of the 16-bit number array seemed unsuitable. Array elements stored in this atypical fashion is not very useful, and such convention (neither big nor little endian) appears not to work too well for dynamic arrays.