304−426A Microprocessor Systems Fall 2009

Lab 1: Assembly and Embedded C

Objective
This exercise introduces the Texas Instrument MSP430 assembly language, the concept of the calling convention and different addressing modes (register, indexed, absolute addressing, etc), as well as the use of embedded C programming. It will also introduce you to the CrossWorks cross-compiler and MSP430 core simulator.

The lab consists of two components:
Part A: Assembly language exercise
Part B: Embedded C exercise

Background

Calling convention

In assembly, parameters for a subroutine are passed on the memory stack and in registers. In the MSP430, the scratch registers are R15:R12. Parameters are placed in these registers in reverse order (i.e. the first parameter in the C-function is placed in R15). If the parameters require more than 16 bits, then multiple registers are used. If there are no free scratch registers, or the parameter requires more registers than remain, then the parameter is pushed onto the stack. For comprehensive examples, do a search for “calling convention” in CrossWorks.

This particular order of passing parameters is a convention by CrossWorks; it is not mandatory and may differ among vendors of compilers.

CrossWorks compiler

When compiling your assembly code, CrossWorks will generate .hzo and .hzx files. The former is your object file and the latter is the image file. The object file is used by the simulator. The image file can be converted to a specific format supported by different processors. For example, to prepare a Texas Instruments hex format:

(1) Select project in Project Explorer.
(2) In the Linker Options group set Additional Output Format to hex.

CrossWorks simulator

The CrossWorks simulator tries to model the processes that take place on the target chip.
A quick start to simulate compiled or linked files:

- Choose Target, then select ‘3 Connect MSP430 Core Simulator’.
- To run the simulation, choose Debug, then select ‘Start Debugging’.

You will be able to inspect memory, program and modify registers while in simulation mode.

Part A

Exercise

Write a subroutine in MSP430 assembly language that adds two seven-digit signed *Binary Coded Decimal* (BCD) numbers.

Your subroutine should expect R15 and R14 to contain the *addresses* of the BCD numbers to be added. R14 should contain the address where the result of the subroutine should be written.

BCD Encoding

Binary Coded Decimal (BCD) data format represents decimal numbers using one nibble (4 bits) per digit. An unsigned 7-digit decimal number may be encoded in BCD format by 28 bits. For signed numbers, an additional sign bit is required. Since it is best to align the data to word boundaries, 32 bits will be used to accommodate the 29-bit long number. We can use any one of the 3 leftover bits as an overflow/underflow indication flag bit.

A 32-bit representation example is:

- Bit 31 (MSB) is the sign bit: '0' for '+value', '1' for '-value',
- Bit 30 is the overflow flag: '0' for 'not overflown', '1' for 'overflown',
- Bit 29,28 are don't care bits,
- Bit 27-0 are data magnitude bits.

So, for example, using 5-digit BCD numbers:

Decimal Number	BCD (binary)	BCD (hexdecimal)
25331	0000 0010 0101 0011 0011 0001	'$025331'
-90814	1000 1001 0000 1000 0001 0100	'$890814'
-45873 (overflown)	1100 0100 0101 1000 0111 0011	'$C45873'
-645873 (overflown)	1100 0100 0101 1000 0111 0011	'$C45873'

Please note that you will use two more digits, than with the above examples.

As a first step, explore the processor instructions that can help in designing efficient BCD arithmetic.

Function Requirements

1. All registers' contents and the stack position should be unaffected by the subroutine call upon returning.
2. The calling convention will obey that of the C compiler. Input registers contain pointers to 16-bit words.
3. The subroutine should leave operands untouched when it returns.
4. The subroutine should also leave the first operand *a1 (pointed to by address a1) unchanged, and replace the second operand with the sum.
5. The subroutine should be location independent. It should be able to run properly when it is placed in different memory locations.
6. The subroutine should not use any global variables. If necessary, use indexed addressing of stack positions.
7. The subroutine should be as fast as possible, but robust.

Test Samples

Be prepared to use your or TA-provided test samples exercising all the interesting cases.

Demonstration

The demonstration involves showing your source code and demonstrating a working program. Your program will be placed at an arbitrary memory location. You should be able to call your subroutine several times consecutively. You should understand what every line in your code does – there will be questions in the demo. This includes the lines in the skeleton; ask if you do not know!

Part B

Exercise

In this part of the experiment, you will implement functions in embedded C that perform simple filtering on a block of data. In the exercise, the input samples will have a fixed-point representation that will re-use the BCD encoding of Part A.

Finite Impulse Response (FIR) filter

A Finite Impulse Response filter is a digital filter which has a finite response to a finite input. Since its internal structure does not contain feedback elements, once the input is removed (set to zero), the filter's response ends when its memory elements are emptied. In the following notation, the output of the filter is given by y[n] and the input given by the samples x[n]. The coefficients of the filter are given by b_i . The equation of the FIR filter is:

(1) $y[n] = b_0x[n] + b_1x[n-1]+...+b_Nx[n-N]$

For example, a small averaging filter would be:

$y[n] = 1/3\ x[n] + 1/3\ x[n-1] + 1/3\ x[n-2]$

The coefficients for this filter would be [0.333..., 0.333..., 0.333...].

FIR filters are quite useful in digital signal conditioning. It is possible, for example, to design FIR filters to remove a certain frequency of noise (notch filter) or do a low-pass or high-pass filter.

We will use a representation of the coefficients that takes into account the decimal position. We will multiply coefficients by 1000. Therefore, a coefficient of 0.333 would be represented as 333. The number N in the filter equation determines the order of the filter. In this experiment, we will keep the order below 10 to save memory. Thus we will be able to implement useful filters while keeping the processing overhead manageable.

Suppose we have the following coefficients implementing a simple 10th order low-pass filter:
[-0.0449, -0.0650, -0.0077, 0.1190, 0.2498, 0.3052, 0.2498, 0.1190, -0.0077,-0.0650, -0.0449]
where the leftmost coefficient is b_0 and the rightmost is b_{10}.

Those coefficients will be represented in our BCD form with the implicit multiplication by 1000 as:
[-45,-65,-8,119, 250, 305, 119, -8, -65, -45] You will notice that the numbers have been rounded.

Now lets consider the input to the system. If we make a 1V input signal correspond to 1000 in our encoding, we would be able to input signals of up to about 1V without risking an overflow in the filter.

If we use equation 1, we see that y[n] will be scaled by 1000 due to the coefficients and that the result will be in millivolts (because we said that 1000 = 1V).

Lets try the simple averaging filter with the scaling applied (all the coefficients are 0.333). Assume the input is 0.5V and then shifts suddenly to 0.7V. A sampling of the input values would be [0.5, 0.5, 0.5, 0.7, 0.7, 0.7]. Scaled up those will become [500,500,500,700,700,700].
Computing the outputs:
y[2] = 333*x[2] + 333*x[1] + 333*x[0] = 333*500 + 333*500 + 333*500 = 499500
y[3] = 333*x[3] + 333*x[2] + 333*x[1] = 333*700 + 333*500 + 333*500 = 566100
y[4] = 333*x[4] + 333*x[3] + 333*x[2] = 333*700 + 333*700 + 333*500 = 632700
y[5] = 333*x[5] + 333*x[4] + 333*x[3] = 333*700 + 333*700 + 333*700 = 699300

The resulting output needs to be divided by 1000*1000 to bring back the decimal point to the correct position.

Thus, the output of the filter (if brought back in volts) would be : [0.4995V, 0.5661V, 0.6327V, 0.6993V].

As you can see, the abrupt step in the input has been smoothed out by the filtering process. Note that for **each** sample produced by the filter, N+1 BCD multiplications and N+1 BCD additions are needed.

With the above conventions, its now possible to design a function that will take a series of samples and perform a digital filter on them to produce a filtered response.

Function Requirements

Design the following C functions:

```
void bcdmult( bcd32_t* arg_a, bcd32_t* arg_b, bcd32_t* result)
```
Where arg_a and arg_b are the locations of the inputs and result is the address where the output should be written.

The "bcdmult" function should call the "bcdadd" routine (written in assembly) repeatedly to perform the multiplication. This function should be tested to see that it can multiply bcd32_t numbers properly and cover corner cases before proceeding to coding the "fir" function.

```
void fir(unsigned int filterOrder, bcd32_t* coeffs, bcd32_t*
inputSamples, bcd32_t* outputValue)
```
Here: -FilterOrder is the order of the filter
-coeffs is a pointer to a table of bcd32_t numbers. The first bcd32_t in the table is b0, the second is b1 and so on.
-inputSamples is a pointer in a table of bcd32_t numbers. Those numbers represent the

input samples to the filter.
-outputValue is a pointer to a free bcd32_t location that will hold the resulting output sample.

1. Use the Crossworks calling conventions for later inclusion into a C program.
2. Upon returning from subroutines, the registers and stack should be untouched.
3. The subroutines should be location independent.

Useful Notes

You may find it useful to separate your functions into separate files. This will definitely be the case for later experiments. The Assembler will be used to assemble each assembly file into a location independent object (.hzo) file. The *Linker* will then link these individual object files to produce a single absolute file (.hzx).

For more information refer to help contents (**CrossStudio for MSP430-> C compiler reference -> assembly language interface).** Also, you can consult application notes, such as **Mixing C and assembler with the MSP430** that is available on WebCT. Although the concepts are the same, please keep in mind that the calling convention and syntax in that document are that of the IAR compiler.

Some C code has been posted on WebCT to help you with this assignment. Please have a look at it. The code is written to be compiled and run on a PC with an ordinary C compiler. However, if the “printf” statements are removed, the same code should also work on CrossStudio for MSP430.

The Linker

In large scale projects, where large programs are usually constructed by teams of programmers, it is necessary to separate the work into smaller units. These smaller pieces of code are composed and tested individually, and are joined together by the Linker program.

When working with these individual pieces of code, it is important to keep them location independent. By postponing the decisions of memory allocation to later times, collisions in memory spaces among different programs can be avoided. Memory for code, data and stack will be allocated when the Linker generates the absolute file (.hzx).

Code Sample

```
/*
// A Main Program to call the subroutines
/*

#include <msp430x14x.h>
#include "bcd.h" // declare bdc_t – could be just unsigned long or alike



void main(void){  // main neither has arguments nor returns anything

// Test code here
```

```
}

// Comments explaining your bcdmult function
void bcdmult( bcd32_t* arg_a, bcd32_t* arg_b, bcd32_t* result){
  // Local variables

  // Code here
}

// Comments explain your fir function.
void fir(unsigned int filterOrder, bcd32_t* coeffs, bcd32_t* inputSamples, bcd32_t*
outputValue) {
  // Your code here...
}

// other functions come here, if needed
// other parameters interpreted as other data types
void other_functions((/*parameters … */) { // Your code

}

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ (New file)
;bcd.h
;
// You must include substantial, clear comments in all header files
// define all the types and routines that C program will use
// your comments here: functionality, inputs, outputs, error conditions
extern void bcdadd(int *c1, int *c2);  // function is coded in assembly

// your comments
void bcdmult( bcd32_t* arg_a, bcd32_t* arg_b, bcd32_t* result);
void fir(unsigned int filterOrder, bcd32_t* coeffs, bcd32_t* inputSamples, bcd32_t*
outputValue);

// your comments
void other_functions();// your code

+++++++++++++++++++++++++++++++++++++++++++++++++
;bcdadd.s43

; comments, purpose of code, inputs, outputs, errors.

     PUBLIC  bcdadd                  ; Declare symbol to be exported
     RSEG    CODE                    ; Code is relocatable
_bcdadd
                                      ;your BCD ADD assembly routine here (Part A)
     ret

     END
```

Demonstration

The demonstration includes showing your source code and demonstrating a working program. Your program should be capable of handling a variety of test cases and should flag errors appropriately.

Report

The report should concisely explain your solution to the problem given, including the bcdadd assembly routine of Part A. You should also explain with illustrations the way the subroutines are linked together and the memory organization that you used. All code should be well-documented.