HOMEWORK ASSIGNMENT #3

Due Wednesday, October 14th, 2009

Remember to follow the Documentation Standards for all code! Use the sample uVision3 project as the starting point for all coding questions. All source code files shall be named in the format "teamX_restofname.s" – change X to your homework team's number.

1. (10 points) Branches versus Conditional Execution

Consider the operations described by

```
if ((R1) is less than (R0)) begin
        (R1) ← 1's-complement of (R1)
        (R0) ← (R1) * 2.5
        end
else if ((R0) is equal to (R1)) begin
        (R1) ← 2's-complement of (R1)
        (R0) ← 0x8100FF00;
        end
else begin
        (R1) ← (R1)^2 + (R0)
        (R0) ← 510;
        end
```

Note that the begin/end pairs demarcate the code that is conditioned on the preceding if/else. Write two code fragments efficiently implementing the behavior. The first one is required to use unconditional instructions except branches may be conditional. The second can use conditional instructions, but no branches at all. Assume that R1 and R0 have signed values in the conditions. Do not use any load instructions or pseudo-instructions in either code fragment. Submit your source code, and answer the following questions.

A. What are the relative advantages/disadvantages of the two approaches?
B. What sorts of the situations would clearly favor one or the other approach.

Although you are only submitting these code fragments, it would be prudent for you to incorporate them into a complete program so you can assemble and test your code.

2. (10 points) Assembly Language Coding

Write code fragments that efficiently perform the following operations.

1. Count the number of bit positions in R0, R1, and R2 that have the same value, and store the count in R3.
2. Search for the bit pattern 10011101 in R0. Consider the bits of the register to be circular (i.e. D31 follows D0), such that the pattern would be found when the register has the value **101**1 1010 10101010 10101010 101**1 0011**. If the pattern exists, set R1=1, otherwise set R1=0. Stop the search as soon as the first instance is found.

Although you are only submitting code fragments, it would be prudent for you to incorporate them into a complete program so you can assemble and test your code.

3. (10 points) Assembly Language Coding

You are to write a subroutine named **umax** (in a file named **teamX_umax.s**) that will determine the maximum value of a variable number of unsigned byte elements. Your procedure will be passed the required parameters as listed in the header below;

```
; Name:         umax
; Description:  This procedure finds the maximum value in a
;               variable length data structure. The starting address
;               of the data structure is contained in R7. The
;               first halfword of the structure is the number of
;               elements in the structure, followed by the data. Each
;               data element is a unsigned byte.
;               The number of elements may be 0, in that case, return
;               the smallest possible unsigned byte value.
; Assumes:      R7 = starting address of the data structure
; Returns:      R0 = maximum value found
; Modifies:     None
umax
```

Your subroutine may not disturb any of the caller's registers. You will want to write a driver program in *main.s* that tests your subroutine – that's how we'll test it. Be sure to EXPORT your subroutine name. A sample of how the data structure should be defined 1s shown below.

```
MyVar    DCW  3
         DCB  1, 2, 3
```

Important: Submit ONLY your source code file **teamX_umax.s** using the dropbox in Learn@UW. Also, submit a **paper copy** of **teamX_umax.s** with the rest of the assignment.

4. (15 points) Implementing the memmove() function with a stack frame

Write a subroutine that functions like the C language ***memmove*** function – one online reference is http://www.cplusplus.com/reference/clibrary/cstring/memmove/ . Your subroutine will copy a block of values from one memory region to another. You must correctly handle any copy specified, including those with overlapping memory blocks.

You are to write a subroutine named ***memmove*** (in a file named **teamX_memmove.s**) that implements memmove. Your subroutine will use stack passing for the arguments, with the assumption that they were pushed in right-to-left order (generally the C convention). This means that the number of characters **num** is pushed first, the **source** address is pushed next, and the **destination** address is pushed last. Set up a standard stack frame in your subroutine to access the parameters. You must assume that the caller has adequate memory allocated at the source and destination.

Return the destination address in R0. You may not disturb any of the caller's registers except R0. You will want to write a driver program in *main.s* that tests your subroutine with a reasonable number of test cases – that's how we'll test it. Be sure to EXPORT your subroutine name. Your

code should be reasonably efficient – the intent is not to have a provably minimal implementation, but deductions will be made for inefficient code.

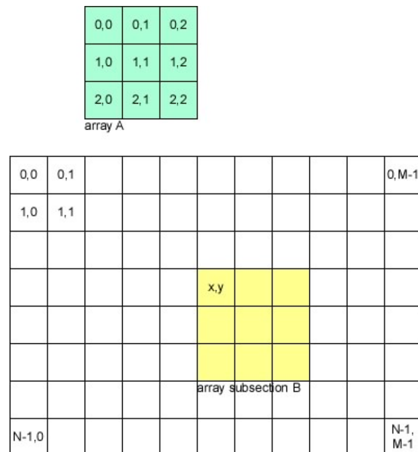
Important: Submit ONLY your program source code file **teamX_memmove.s** using the dropbox in Learn@UW. Also, submit a **paper copy** of **teamX_memmove.s** with the rest of the assignment.

5. (20 points) 2D Array Manipulation

A common operation in image processing is the computation of the sum-of-absolute-differences (SAD) between the elements in two arrays. Assume that there are two 2-dimensional arrays named A and B, where A and B are the same size. If A and B are M-by-N arrays (where M is the number of elements in a row and N is the number of rows), the SAD is computed as

$$SAD = \sum_{j=0}^{j<N} \sum_{i=0}^{i<M} |A_{j,i} - B_{j,i}|$$

The SAD is used as a measure of how similar two areas in an image are. In particular, we are often very interested in trying to figure out if an area in one frame of a video image has moved to another location in the next frame. In that case, you compute the SAD between a small array (A) and a subsection (B) of a larger array, as shown below. This is the calculation that you will write a subroutine to perform.



Write a subroutine **compSAD** that will compute the SAD between a 3x3 array and a 3x3 subsection of a larger array. The subroutine will be passed the following parameters using register passing;

R7	base address of 3x3 array
R8	base address of larger array
R9	row length of larger array
R10	x coordinate of upper left corner of subsection of larger array
R11	y coordinate of upper left corner of subsection of larger array

Return the computed SAD in RO.

Assume that all arrays will be stored in memory by rows (i.e. element (0,0) first, followed by (0,1) through (0,X-1), then (1,0) through (1,X-1), etc.). Assume that the subroutine is always called with valid arguments. Assume unsigned byte arrays.

All of your subroutine code is to be in a file named **teamX_compsad.s**. Your subroutine may not modify any of the caller's registers except R0. Do **not** declare a data area in your **teamX_compsad.s** file. If you need temporary storage, allocate it on the stack. Do not place any other code in this file; place your driver code for testing your subroutine in a separate file that is not submitted. Be sure to EXPORT your subroutine name. We will use a separate driver file that links to your subroutine to test it and verify its accuracy – you should do the same before submitting your work. A sample source file with a set of array definitions is available on the course web page. If you were to run your subroutine using these arrays with R10=3 and R11=2, the SAD should be 0. Be sure to test your subroutine thoroughly.

Your code should be reasonably efficient – the intent is not to have a provably minimal implementation, but deductions will be made for significant inefficiencies. Take advantage of the addressing modes that ARM provides.

Important: Submit ONLY your subroutine source code file **teamX_ compsad.s** using the dropbox in Learn@UW. Also, submit a **paper copy** of **teamX_ compsad.s** with the rest of the assignment.

6. (5 points) ARM7TDMI Stack Initialization

Review the initialization code in the file ***aduc7026.s*** in the sample uVision3 project.

1. Describe what the reset handler code (after the comment "***; Setup Stack for each mode***") accomplishes, and draw a memory map showing where the various stack pointers point and where the stack areas are for each of the modes.
2. Some modes have a zero length stack? Is this okay? Can we call a subroutine from that mode?

7. (20 points) Recursive Programming

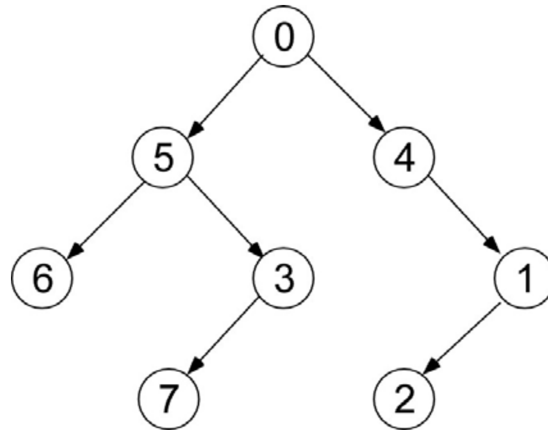
A *binary tree* is a type of hierarchical data structure where each node has at most two child nodes. The *root* node of the tree is the one node that has no parents - it is the starting point of the tree. Every other node on the tree has only a single parent node. Note that this similar to a single-linked list, except there can be two *next* nodes.

A common operation we need to perform is to *traverse* the tree, which means to visit each node on the tree once. This operation can be implemented recursively by a subroutine that is passed the root node of a subtree (a tree's root node or any node within the tree) and then calls itself twice to visit the child nodes of the subtree. We will start the traversal process by calling this subroutine and passing it the root node of the tree. The termination condition for the recursion is a node that has no children.

The data that comprises the tree is a series of node data structures allocated in memory. The root node of the tree is named ***node0***. Each node is represented by three data words. The first word is the number of the node, the second word is the address of the left child node, and the third word is the address of the right child node. If a node does not have a child node on a branch, then the value -1 (0xFFFFFFFF) is stored as the address. An example data structure is shown below;

```
                ;node       left child      right child
node0  DCD   0,          node5,          node4
node1  DCD   1,          node2,          -1
node2  DCD   2,          -1,             -1
node3  DCD   3,          node7,          -1
node4  DCD   4,          -1,             node1
node5  DCD   5,          node6,          node3
node6  DCD   6,          -1,             -1
node7  DCD   7,          -1,             -1
```

This data would be interpreted as the tree shown below.



You are to write a subroutine ***traverse*** to recursively traverse a binary tree. The subroutine will be passed the address of the root node of the subtree it is to traverse. The subroutine will write the number of that node to SRAM (as a **byte**) to the address specified by R10, and then will recursively traverse the left and right sub-trees below it (taking the left branch first). (Be sure to increment the R10 after writing data, so the next call of ***traverse*** will not overwrite the data. R10 should NOT be saved and restored by your subroutine.) If a sub-tree is empty, do not attempt to traverse that sub-tree.

- Input parameters
  - : **root node address (on stack)**
  - : **R10 – SRAM address to write node number to**
- Modifies: **R10**
- Returns: **none**

If your subroutine were passed the address of ***node0*** as illustrated above, then it should write to SRAM (using R10) as follows;

    0x00010000 00 05 06 03 07 04 01 02

You may assume that the data structure your subroutine operates on will always be valid, i.e. no cyclic paths, no invalid addresses, etc.

Place your subroutine source code in a file named **teamX_traverse.s**. Your subroutine may not modify any of the caller's registers except R10. Do **not** declare or use a data area in your subroutine file. Do **NOT** place any other code in this file. We will use a driver file that links to your subroutine to completely test your code and verify its correctness – you should do the same before submitting your work. A sample driver file is posted on the course web page. Your code should be reasonably efficient – the intent is not to have a provably minimal implementation, but deductions will be made for significant inefficiencies.

Important: Submit ONLY your subroutine source code file **teamX_traverse.s** using the dropbox in Learn@UW. Also, submit a **paper copy** of **teamX_traverse.s** with the rest of the assignment.

8. (10 points) Exam Question

Design one original quiz question operating at Bloom's Taxonomy level 3 for any material covered in Module 3. This must test one of the educational objectives (see http://eceserv0.ece.wisc.edu/~morrow/ECE353/objectives.pdf) in a specific problem. Explicitly state which particular educational objective you are attempting to test (i.e. paste a copy of the objective into your document). Provide a complete, detailed solution to your question.