

HOMEWORK ASSIGNMENT #3Due Friday, March 6th, 2009

Remember to follow the Documentation Standards for all code! Use the sample Keil project as the starting point for all coding questions. All source code files shall that are submitted via Learn should be named in the format “teamX_<funcName>.s” – where X is your team number and <funcName> is the name of the function given in the problem.

1. (10 points) Branches versus Conditional Execution

Consider the operations described by

```
if ((R0) is less than (R1)) begin
    (R2) ← 2's-complement of R5
    (R3) ← (R4) * 32768
end
else if ((R1) is less than (R0)) begin
    (R2) ← 1's-complement of R5
    (R3) ← 0;
end
else begin
    (R2) ← (R5)2 + (R6)
    (R3) ← 0x10000008;
end
```

Consider R0 and R1 to contain unsigned numbers.

Note that the begin/end pairs demarcate the code that is conditioned on the preceding if/else.

- Write a code fragment efficiently implementing the behavior using no branches (conditional execution only).
- Write a code fragment efficiently implementing the behavior using unconditional instructions except for branches.
- What are the relative advantages/disadvantages of the two approaches?
- What sorts of situations would clearly favor one approach over the other?

Although you are only submitting this code fragment in paper form (not in drop box), it would still be prudent for you to incorporate it into a complete program so you can assemble and test your code.

2. (10 points) Assembly Language CodingWrite **routines** that efficiently perform the following operations:

- Count the number of leading zeros (starting at the MSB) in R0 and returns that number in R1. For example if R0 contained 0x0623ABCD the result would be 5 stored in R1.
- Search for the bit pattern 000110010 in R4. 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 **0010** 1010 10101010 10101010 1010 **0011**. If the pattern exists, return R5=1, otherwise return R5=0. Stop the search when exhausted all possible search positions, or when the first match is found.

Remember to either not alter any user state (registers) or explicitly comment what registers are modified in the header of your routines. Although you are only submitting these routines in paper form (not in dropbox), it would still be prudent for you to incorporate them into a complete program so you can assemble and test your code.

3. (15 points) LUTs & Context save/restore

You are to write a subroutine named **ln1K** (in a file named **teamX_ln1K.s**) that will return 1000 times the natural log (ln) of the 4-bit argument passed in R7. The answer returned should be placed in R0. Don't knock yourself out with numerical methods, use a LUT instead. For invalid results return 0xFFFFFFFF.

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.

Important: Submit ONLY your source code file **teamX_ln1K.s** using the dropbox in Learn@UW. Also, submit a **paper copy** of **teamX_ln1K.s** with the rest of the assignment.

4. (20 points) Implementing a merge sort routine using a stack frame.

Write a subroutine that combines two sorted arrays of signed 16-bit (halfwords) into a single sorted array. Each of the two arrays passed in were sorted smallest to largest. The resulting array should also be sorted smallest to largest. The caller will push arguments (in this order):

- 1) A word that tells the number of entries in the array0
- 2) A word that points to (is the address of) the 1st element in array0
- 3) A word that tells the number of entries in the array1
- 4) A word that points to (is the address of) the 1st element in array1
- 5) A word that points to the address where they want the ordered results stored. (**NOTE:** This address could have an overlap, or be the same as either of the arrays passed in)

You are to write a subroutine named **merge_hwrds** (in a file named **teamX_merge_hwrds.s**) that uses stack frame passing as covered in week6 lecture. The length of each array passed in will always be at least 1 entry.

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_merge_hwrds.s** using the dropbox in Learn@UW. Also, submit a **paper copy** of **teamX_merge_hwrds.s** with the rest of the assignment.

5. (10 points) Flow Chart

Using the conventions for drawing flowcharts discussed in class draw a flowchart for the **sort_hwrds** subroutine given in Problem 4. Neatness & completeness matter.

6. (15 points) Recursive Calculation of Factorials

Computing factorials using recursion is perhaps about the most memory inefficient, and speed inefficient way it can be done,- not to mention unnecessarily complex. However, use of recursion is one of the best ways to prove how smart you are (and in our case how well you understand the stack frame concept). You are to write a subroutine to recursively calculate the factorial of any number whose result would not overflow a 32-bit unsigned number (within the limits of the caller's stack, of course – but that is not a problem for your subroutine to worry about).

Any factorial can be calculated recursively as $n! = n \cdot (n-1)!$ for $n \geq 1$, with $0! = 1$. Your subroutine **must perform this calculation recursively** for an arbitrary n using stack parameter passing and have an interface as follows:

- Subroutine name: **recur_fact**
- Input parameters: **n - single word on stack (interpreted as unsigned)**
- Return: **R0 = result**

Your subroutine should not have any loops, but should correctly call itself using stack parameter passing to implement the recursion. The recursion should terminate when the subroutine is called with a passed value of 0. Place your subroutine source code in a file named **teamX_recur_fact.s**. Your subroutine may not modify any of the caller's registers except R0, and may NOT use any memory variables. Do **not** declare a data area in your **teamX_recur_fact.s** file. Do **NOT** place any other code in this file; place driver code for testing your subroutine in a separate file that is not submitted. We will use a separate driver file that links to your subroutine to completely test your procedure and verify its accuracy – you should do the same before submitting your work.

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

7. (10 points) ARM7TDMI Stack Initialization

Review the initialization code in the file **aduc7026.s** in the sample Keil 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?

8. (10 points) Exam Question

Design one original quiz question operating at Bloom's Taxonomy level 3 for any material covered in Module 3. Provide a complete, detailed solution to your question.