

HOMEWORK ASSIGNMENT #3 SOLUTIONDue 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.

- a) Write a code fragment efficiently implementing the behavior using no branches (conditional execution only).

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Name: prob1_cond
;; Description: Does routine in prob1 HW3 using only
;;              conditional execution (no branches)
;; Author: Eric Hoffman
;;
;; Assumes: R0 & R1 contain numbers to be compared
;; Modifies: R2 & R3 (returned results)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
prob1_cond
    PUSH    {R8, LR}    ; Save of context an LR
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    CMP     R0, R1      ; compare and set flags
    RSBLO  R2, R5, #0   ; Put 2's comp of R5 in R2 if R0<R1
    MOV     R8, #32768   ; load R8 with constant
    MULLO  R3, R4, R8   ; R3 gets R4*32768 if R0<R1
    MVNHI  R2, R5       ; R2 gets R5 inverted if R1<R0
    MOVHI  R3, #0       ; R3 gets zero if R1<R0
    MLAEQ  R2, R5, R5, R6 ; R2 gets R5 squared if R1=R0
    MOVEQ  R3, #0x10000008 ; R3 gets 0x10000008 if R1=R0
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    POP     {R8, PC}    ; restore context and return

```

- b) Write a code fragment efficiently implementing the behavior using unconditional instructions except for branches.

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Name: probl_branch
;; Description: Does routine in probl HW3 using unconditional
;;              instructions with branches
;; Author: Eric Hoffman
;;
;; Assumes: R0 & R1 contain numbers to be compared
;; Modifies: R2 & R3 (returned results)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
probl_branch
    PUSH    {R8, LR}    ; Save of context an LR
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    CMP     R0, R1      ; compare and set flags
    BHS    nxt_tst     ; if not strictly lower then perform next test
    RSB    R2, R5, #0   ; Put 2's comp of R5 in R2 if R0<R1
    MOV     R8, #32768  ; load R8 with constant
    MUL    R3, R4, R8   ; R3 gets R4*32768 if R0<R1
    B      return
nxt_tst  BEQ    eq_case ; if not equal then R
    MVN    R2, R5      ; R2 gets R5 inverted if R1<R0
    MOV    R3, #0      ; R3 gets zero if R1<R0
    B      return
eq_case  MLA    R2, R5, R5, R6 ; R2 gets R5 squared if R1=R0 |
    MOV    R3, #0x10000008 ; R3 gets 0x10000008 if R1=R0
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
return   POP     {R8, PC} ; restore context and return

```

c) What are the relative advantages/disadvantages of the two approaches?

Conditional execution of an instruction can result in a smaller number of instructions in the code by eliminating branches. However, the execution time of the code may be higher if there are more conditional instructions that are not executed as compared to the time required to do the branches.

One could also argue that the conditional execution is harder to follow since you must pay strict attention to the condition mnemonics.

d) What sorts of situations would clearly favor one approach over the other?

Conditional execution will generally be best for short segments of code that executes based on a single condition, since you don't need the branch instruction, thus saving one instruction each time you need to do conditional execution. (This also prevents a pipeline flush, which is increasingly more serious as the processor pipeline is made deeper.) More complicated tasks that require selecting among two or more options for execution will still require using branches in most cases. The most efficient method will depend on the specifics of the code.

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 Coding

Write **routines** that efficiently perform the following operations:

1. 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.

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Name: prob2_leadzeros
;; Description: returns a count of the leading zeros of R0
;;              in register R1
;;
;; Author: Eric Hoffman
;;
;; Assumes: R0 data to have leading zeros counted
;; Modifies: R1 (returned result)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
prob2_leadzeros
        PUSH    {R0,R8,LR}      ; Save of context an LR
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        MOV     R8, #0x01       ; going to need a register to set LSB as we shift
        MOV     R1, #0          ; start it at count of zero
lz_lp   TST     R0, #0x80000000  ; check the MSB = 0
        BNE    return_lz       ; we are done, there is a 1 in MSB
        ADD    R1, R1, #1       ; increment count
        ORR    R0, R8, R0, LSL #1 ; Shift R0 left and set LSB
        B      lz_lp           ; keep checking
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
return_lz POP    {R0,R8,PC}      ; restore context and return

```

2. 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.

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Name: prob2_pattsearch
;; Description: looks for pattern 000110010 in R4. If
;;              pattern found it return R5=1, else R5=0
;; Author: Eric Hoffman
;;
;; Assumes: R4 = register to have pattern searched
;; Modifies: R5 (returned results)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
patt EQU 0x32 ; define pattern we are looking for
prob2_pattsearch
        PUSH    {R4,R8,R9,R10,LR} ; Save of context an LR
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        LDR    R10, =(0x1FF) ; need this mask in a register
        MOV    R5, #0 ; assume not found
        MOV    R8, #31 ; R8 will be loop counter, there are 31 possible positions for target
nxt_srch AND    R9, R4, R10 ; mask off to look in lower 9-bits
        CMP    R9, #patt ; compare against with pattern in LS 9-bits
        MOVEQ  R5, #1 ; found so set R5
        BEQ    rtn_srch ; if found we are done
        ROR    R4, #1 ; rotate the search register
        SUBS  R8, R8, #1 ; decrement loop counter
        BNE    nxt_srch
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
rtn_srch POP    {R4,R8,R9,R10,PC}

```

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.

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Name: ln1K
;; Description: returns (in R0) 1000*ln(R7). R7 should be
;;             range checked for a 4-bit number. Return a
;;             0xFFFFFFFF for invalid arguments
;;
;; Author: Eric Hoffman
;;
;; Assumes: R7 value to return 1000*ln
;; Modifies: R0 (returned result)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
ln1K
        PUSH    {R8,LR}      ; Save of context an LR
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        CMP     R7, #15      ; compare to largest valid 4-bit value
        BHS    rtnrtn_invl   ; return an invalid result
        LDR     R8, =(tbl_addr)
        LDR     R0, [R8, R7, LSL #2] ; grab result from table
rtnrtn_invl MOVHS    R0, #0xFFFFFFFF ; return invalid result (only if out of range)
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
return_ln  POP     {R8,PC}    ; restore context and return

        ;;;;;;;;;;;;;;;;;; ln table follows ;;;;;;;;;;;;;;;;;;
tbl_addr  DCD    0xFFFFFFFF, 0x0, 693, 1099, 1386, 1609, 1792, 1946,\
                2079, 2197, 2303, 2398, 2485, 2565, 2639, 2708

```

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.

[Take this Link to one possible solution](#)

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.

This problem was just given as punishment for taking this class... We will be looking to see that you made proper use of the symbols, and that you kept the flowchart at a high enough level of abstraction to be useful.

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.

```

////////////////////////////////////
;; Name: recur_fact
;; Description: Passed a single argument on the stack. Returns
;;              the factorial of that argument in R0. Must be
;;              implemented using recursion to prove you worth
;;              as a human
;;
;; Author: Eric Hoffman
;;
;; Assumes: 1 argument passed on the stack
;; Modifies: R1 (return result)
////////////////////////////////////
recur_fact
    PUSH    {R1,R11,LR}      ; Save frame pointer (R11) and LR
    //////////////////////////////////////
    MOV     FP, SP           ; set frame pointer to current SP
    LDR     R1, [FP, #12]    ; grab the input argument
    MOVS    R1, R1           ; set the z flag looking for zero
    MOVEQ   R0, #1          ; Zero factorial is 1
    BEQ     fact_done        ; return with zero
    ////////////////////////////////////// else we have recursive call to do //////////////////////////////////////
    SUB     R1, R1, #1
    PUSH    {R1}            ; push argument for a new call to recur_fact
    BL      recur_fact
    ADD     R1, R1, #1       ; restor R1 = R1 + 1 before call
    MUL     R0, R0, R1       ; R0 = n*(n-1)!
    ADD     SP, SP, #4       ; deallocate arument to recursive call
    //////////////////////////////////////
fact_done POP     {R1,R11,PC} ; restore context and return

```

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.

- 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.
- Some modes have a zero length stack? Is this okay? Can we call a subroutine from that mode?

The reset handler is the first code to execute after the processor initially starts running after power is applied or whenever a reset occurs for any reason. The interrupt vector table is hard-coded into the start-up code. The reset handler initializes the stacks for each operating mode by setting their stack pointers to the proper locations in memory. The stack area memory map is shown below:

	Undefined Instruction (UND) – NO SPACE ALLOCATED Abort (ABT) – NO SPACE ALLOCATED
0x104EF 0x104B0	64 bytes - Fast Interrupt (FIQ)
0x104AF 0x10470	64 bytes - Interrupt (IRQ)
0x1046F	8 bytes - Supervisor (SVC)

0x10468	
0x10467	
0x10068	1024 bytes - User (USR)

The stack is set up as a full descending (FD) topology, so the stack pointers point one location above the top of each area. Note that the actual starting address of the stack memory space is determined by the linker, and can vary depending on the link order and what other data is allocated.

Some modes have a zero length stack? Is this okay? Can we call a subroutine from that mode?

If we can guarantee that we won't ever use the stack in a given mode, then we can get away with out allocating any stack space for it. If we inadvertently used the stack in a mode without space allocated for its stack, it would overwrite the stack of the mode below it in the stack area. Note that this is also a potential hazard even if we have allocated stack space for a given mode but end up using more space than allocated when our program runs. In both cases, the stack corruption would affect a different mode's operation – this can be very hard to find and debug.

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.