

HOMEWORK ASSIGNMENT #3-SOLUTION**1. (10 points) Branches versus Conditional Execution**

```

; unconditional instructions with conditional branches
    CMP    R1, R0                ; compare registers
    BLT    r1_less_than_r0
    BEQ    r1_equal_r0
r1_greater_than_r0
    MLA    R2, R1, R1, R0
    MOV    R1, R2                ; R1 <= R1*R1 + R0
    MOV    R0, #255              ; R0 <= 510
    MOV    R0, R0, LSL #1
    B      end_using_branches
r1_less_than_r0
    MVN    R1, R1                ; R1 <= NOT R1
    MOV    R0, R1, LSL #1        ; R0 <= 2*R1
    ADD    R0, R1, ASR #1        ; R0 <= R0 + 0.5*R1
    B      end_using_branches
r1_equal_r0
    RSB    R1, R1, #0            ; R1 <= -R1
    MOV    R0, #0x81000000
    ORR    R0, #0x0000FF00        ; R0 <= 0x8100FF00
end_using_branches

; conditional instructions with no branches
    CMP    R1, R0                ; compare registers
;r1_greater_than_r0
    MLAGT  R2, R1, R1, R0
    MOVGT  R1, R2                ; R1 <= R1*R1 + R0
    MOVGT  R0, #255              ; R0 <= 510
    MOVGT  R0, R0, LSL #1
;r1_less_than_r0
    MVNLT  R1, R1                ; R1 <= NOT R1
    MOVLT  R0, R1, LSL #1        ; R0 <= 2*R1
    ADDLT  R0, R1, ASR #1        ; R0 <= R0 + 0.5*R1
;r1_equal_r0
    RSBEQ  R1, R1, #0            ; R1 <= -R1
    MOVEQ  R0, #0x81000000
    ORREQ  R0, #0x0000FF00        ; R0 <= 0x8100FF00

```

A. 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.

B. 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 is generally better suited to using branches, as will code that calls a subroutine. (We generally cannot assume that flags are preserved by a subroutine call.) The most efficient method will depend on the specifics of the

code. Determining which is best requires getting the time needed to execute an instruction compared to the time needed to execute a branch. If a branch takes x usec to execute and an instruction takes $y=x/2$ usec, then doing a conditional execution of one instruction will only take half as much time as branching around it. The code fragment below tests this idea. It was run on the simulator and the time at each instruction was noted (the time is shown on the status bar at the bottom of the tool window). The time to execute a branch instruction was 1.17 usec. Each conditional instruction not executed took about 0.38 usec. So, if we use less than three conditional instructions, it is faster than a conditional branch. If we use three conditional instructions, it is the same amount of run time. If we use more than three conditional instructions, it is faster to use a conditional branch.

Arm Instruction timing		
Conditional branch vs. conditiona		
line of code	time at start (usec)	time to execute (usec)
1	11.49	1.17
2	12.66	
3a	14.17	0.38
3b	14.55	0.39
3c	14.94	0.38
4	15.32	
	sum	1.15

```

__main
    MOV    R0, #1
    CMP    R0, #1
    BEQ    skip          ;1
    ADD    R1, R0, #10   ;1a
    ADD    R2, R1, #10   ;1b
    ADD    R3, R2, #10   ;1c
skip
    ADD    R1, R0, #5    ;2

    MOV    R0, #1
    CMP    R0, #1       ;3
    ADDNE  R1, R0, #10   ;3a
    ADDNE  R2, R1, #10   ;3b
    ADDNE  R3, R2, #10   ;3c
skip1
    ADD    R1, R0, #5    ;4

    B      __main

```

2. (10 points) Assembly Language Coding

```

MOV      R0, #0x1000000F    ;test case
MOV      R1, #0x30000008
MOV      R2, #0xF000000F

;problem 2.1
EOR      R4, R0, R1        ;R4 has 0s where R0/R1 bits are the same
EOR      R5, R1, R2        ;R5 has 0s where R1/R2 bits are the same
ORR      R4, R4, R5        ;R4 has 0s where R0/R1/R2 bits are the
same
MVN      R4, R4            ;R4 has 1s where R0/R1/R2 bits are the
same
MOV      R5, #32           ;iteration count
MOV      R3, #0            ;clear counter
p2_1_loop
MOVS     R4, R4, ROR #1    ;put LS bit in carry flag
ADC      R3, R3, #0        ;increments count if LS bit was 1
SUBS     R5, R5, #1        ;done with loop?
BNE      p2_1_loop

;problem 2.2
LDR      R0, =(2_1011101010101010101010101010110011) ;test case
MASK EQU 2_11111111
TARGET EQU 2_10011101

MOV      R2, #32           ;iteration count
MOV      R1, #0            ;flag as not found
p2_2_loop
AND      R3, R0, #MASK     ;mask off LS bits
CMP      R3, #TARGET       ;is it the target value?
MOVEQ R1, #1               ;if so, flag as found
BEQ      p2_2_end          ;and exit
MOV      R0, R0, ROR #1    ;rotate to expose next bit area to mask
SUBS     R2, R2, #1
BNE      p2_2_loop
p2_2_end

```

3. (10 points) Assembly Language Coding

```
; Filename:      umax.s
; Author:       ECE 353 staff
; Description:   homework #3 problem #3

        ARM                      ;use ARM instruction set
        EXPORT   umax

        AREA    FLASH, CODE, READONLY

; 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 an 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
        PUSH   {R1-R2, LR}      ;context save
;
        MOV    R0, #0           ;initialize return value
        LDRH   R1, [R7], #2     ;load count and index
umax_loop
        CMP    R1, #0          ;check for 0 count
        BEQ   umax_exit       ;if so, we are done
;
        LDRB   R2, [R7], #1    ;get next data element and index
        CMP    R2, R0          ;is it the new max value?
        MOVHI  R0, R2          ;if so, make it so
        SUB    R1, R1, #1      ;decrement count
        B     umax_loop
;
        MOV    R0, R2          ;save result
        MOV    R1, R3
umax_exit
        POP    {R1-R2, PC}     ;context restore/return

        END
```

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

```

; Filename:      memmove.s
; Author:       ECE 353 staff
; Description:   homework #3 problem #4

        ARM                ;use ARM instruction set
        EXPORT memmove

        AREA FLASH, CODE, READONLY

; Name:         memmove
; Description:  Implementation of C memmove function
; Assumes:     Parameters pushed on stack in order
;              num - number of bytes to copy
;              src - source address
;              dest - destination address
; Returns:     Nothing
; Modifies:    None
memmove
        PUSH {R0-R3, R11, LR}    ;context save
;
        MOV R11, SP              ;set up stack frame
        LDR R0, [R11, #6*4]      ;get destination address
        LDR R1, [R11, #7*4]      ;get source address
        LDR R2, [R11, #8*4]      ;get count
;
        CMP R1, R0               ;check for direction of transfer
        BLO memmove_src_lower
memmove_src_higher               ;source is higher, so start from bottom
        CMP R2, #0               ;are we done?
        BEQ memmove_exit
        LDRB R3, [R1], #1        ;copy data and post-index
        STRB R3, [R0], #1
        SUB R2, R2, #1           ;decrement byte count
        B memmove_src_higher
;
memmove_src_lower               ;source is lower, so start from top
        ADD R0, R2               ;get addresses at top of memory blocks
        ADD R1, R2
memmove_src_lower_loop
        CMP R2, #0               ;are we done?
        BEQ memmove_exit
        LDRB R3, [R1, #-1]!      ;copy data with pre-index
        STRB R3, [R0, #-1]!
        SUB R2, R2, #1           ;decrement byte count
        B memmove_src_lower_loop
;
memmove_exit
        POP {R0-R3, R11, PC}     ;context restore/return

        END

```

5. (20 points) 2D Array Manipulation

```

; Filename:      compsad.s
; Author:       ECE 353 staff
; Description:   homework #3 problem #5

        ARM                ;use ARM instruction set
        EXPORT  compSAD

        AREA  FLASH, CODE, READONLY

; Name:        compSAD
; Description:  This procedure finds the sum-of-absolute_differences
;              (SAD) between a 3x3 array and a 3x3 section of a
;              larger array. The arrays are assumed to be unsigned bytes.
; Assumes:     R7 = base address of the 3x3 array
;              R8 = base address of the larger array
;              R9 = row length of the larger array
;              R10 = lowest x coordinate of 3x3 subsection
;              R11 = lowest y coordinate of 3x3 subsection
; Returns:     R0 = computed SAD
; Modifies:    None
compSAD
        PUSH  {R1-R4, R7-R11, LR}    ;context save
;
        MOV   R0, #0                ;initialize return value
        MLA   R1, R11, R9, R10       ;compute offset (y*row_length)+x
        ADD   R8, R1                 ;and add to large array base address
        SUB   R9, R9, #3             ;increment from (y,x+2) to (y+1,x)
        MOV   R1, #3                ;outer loop counter
compSAD_outer_loop
        MOV   R2, #3                ;inner loop counter
compSAD_inner_loop
        LDRB  R3, [R7], #1          ;load 3x3 array element and index
        LDRB  R4, [R8], #1          ;load larger array element and index
        SUBS  R3, R3, R4            ;get difference
        RSBMI R3, R3, #0            ;negate if negative
        ADD   R0, R0, R3           ;add to running sum
        SUBS  R2, R2, #1            ;decrement inner loop count
        BNE   compSAD_inner_loop
        ADD   R8, R8, R9           ;move pointer to next row of large array
        SUBS  R1, R1, #1            ;decrement outer loop count
        BNE   compSAD_outer_loop
;
compSAD_exit
        POP   {R1-R4, R7-R11, PC}   ;context restore/return

        END

```

6. (5 points) ARM7TDMI Stack Initialization

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
0x10497 0x10458	64 bytes - Fast Interrupt (FIQ)
0x10457 0x10418	64 bytes - Interrupt (IRQ)
0x10417 0x10410	8 bytes - Supervisor (SVC)
0x1040F 0x10010	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 not 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. Calling a subroutine will corrupt the return value stored in R14, so we must have a way to save R14 and restore it after the call (i.e. put it in a memory location or a register). The subroutine must not use the stack in any way.

7. (20 points) Recursive Programming

```

; Filename:      traverse.s
; Author:       ECE 353 staff
; Description:  homework #3 problem #6

        AREA    FLASH, CODE, READONLY
        ARM

        EXPORT  traverse

; Name: traverse
; Description: Binary tree traversal
; Assumes:     Address of root node pushed on stack
;             R10 has address for output data (increment if used)
; Returns:     None
; Modifies:    None
traverse
        PUSH   {R0-R1, R11, LR}      ; context save
;
        MOV    R11, SP                ; set up stack frame
        LDR    R1, [R11, #4*4]        ; get root node address
traverse_write_node
        LDR    R0, [R1, #0]           ; get node number
        STRB   R0, [R10], #1          ; write node number and increment address
traverse_left
        LDR    R0, [R1, #4]           ; get left node address
        CMP    R0, #-1                ; check if branch exists
        BEQ    traverse_right         ; if not, go on to right branch
        PUSH   {R0}                   ; left node address
        BL     traverse               ; recursive call
        ADD    SP, SP, #4             ; clean up stack
traverse_right
        LDR    R0, [R1, #8]           ; get right node address
        CMP    R0, #-1                ; check if branch exists
        BEQ    traverse_exit          ; if not, go to exit
        PUSH   {R0}                   ; left node address
        BL     traverse               ; recursive call
        ADD    SP, SP, #4             ; clean up stack
traverse_exit
        POP    {R0-R1, R11, PC}       ; context restore/return

        END

```