

# EE 3381: Microcontrollers and Embedded Systems

Spring 2014 Second Exam (April 24, 2014)

**Instructions: Read carefully before beginning.**

- The time for this exam is 1 hour and 20 minutes.
- You may only refer to the ARM Instruction Set Quick Reference Card. You may NOT use your textbooks, lecture notes, lab assignments, lab solutions, a computer, or phone.
- During the exam, you may not contact Professor Camp, even for clarifications: Interpret problems as best as you can and explain your interpretations/assumptions.
- Write your answers on the exam pages provided, using front and back if needed.
- This exam is covered by the SMU Honor Code. Please sign acknowledging the following pledge: On my honor, I have neither given nor received any unauthorized aid on this examination.
- Good luck!

Signature: \_\_\_\_\_

Name (Printed): \_\_\_\_\_

Problem	Score	Total Possible
1		12
2		6
3		10
4		11
5		20
6		20
7		21
Total		100

1. (12 pts) Translate each of the following conditions into a single ARM instruction (a condition flag state table is on the last page for your reference for b and c):
  - a. Push a block of registers from r7 to r11 and the link register onto the stack using a full ascending stack convention. *You may not use FA* in the instruction, but should use the appropriate choice out of IA, IB, DA, or DB, representing increment(I)/decrement(D) after(A)/before(B). You may assume that the SP has been initialized properly.
  - b. Branch and link to a label called *MoonPie* that branches if condition code Z is set.
  - c. Add r7 to r8 and put the result in r9 only if condition code N is clear. Also, this addition should update the condition codes according to the result stored in r9.
2. (6 pts) Write the necessary lines of code representing a while loop (*not* a do-while loop) that will execute 10 times.
3. (10 pts) Assume that memory and registers r0 through r3 appear as follows (Note: The addresses are pictured from least to greatest from top to bottom.):

Address	Value	Register	Value
0x7FFC	0xCAFE1234	r0	0x13
0x8000	0x00000001	r1	0xFFFFFFFF
0x8004	0xFEEDDEAF	r2	0xEEEEEEEE
0x8008	0x00008888	r3	0x8008
0x800C	0x12340000	-	-
0x8010	0xBABE0000	-	-

Change the memory and register contents after executing the instruction. *Remember that this is a block copy and the lowest register number takes the first word read from memory whereas the highest register number takes the last word read from memory.*

STMDA    r3!, {r0, r1, r2}

4. (11 pts) Read the following code and describe what it does. Do not simply interpret line for line, but rather come up with the big picture of the operation the code performs.

```
num      AREA    WhoAmI, CODE, READONLY
        EQU     20
        ENTRY

        MOV     r0, #num
        MOV     r0, r0, LSL #2
        ADR     r1, elements
        MOV     r2, r1
        SUB     r2, r2, #4

Func1    ADD     r3, r1, r0
        SUB     r3, r3, #4
        MOV     r4, #0
        ADD     r2, r2, #4

Func2    LDR     r5, [r3], #-4
        LDR     r6, [r3]
        CMP     r5, r6
        BGT     Func3

        STR     r5, [r3], #4
        STR     r6, [r3]
        ADD     r4, r4, #1
        SUB     r3, r3, #4

Func3    CMP     r3, r2
        BNE     Func2
        CMP     r4, #0
        BNE     Func1

Done     B       Done

data     DCD     33, -35, 6, 54, 77, 3, 4, 2, 1, 0
        DCD     -5, 1, 2, 3, 0, 12, 22, 7, 6, 5
        END
```

What does this code do?

5. (20 pts) **Subroutines.** Complete the following code that takes the factorial program discussed earlier in the semester (Chapter 3) into a subroutine, using full descending stacks. *You may not use FD* to push values to the stack and pull values from the stack. You should use the appropriate choice from the following options: IA, IB, DA, DB. Notice that there are two different subroutines that do the same thing, but pass by register in the first and pass by reference in the second.

```

                AREA    FactorialPassRegAndRef, CODE, READONLY
SRAM_BASE EQU    0x40000000
                ENTRY
                LDR     sp, =SRAM_BASE
                LDR     r3, =SRAM_BASE+100

-----
                                ; pass test value by register using r0
-----
                                ; call appropriate function below
                                ; result now in r1
-----
                                ; save parameter in memory
                                ;     (at SRAM_BASE+100)
-----
                                ; call appropriate function,
                                ;     passing r3 as reference
                                ; result now in [r3]

stop          B          stop
FactReg       -----
-----
                                ; save appropriate registers
                                ;     on stack for reentrancy
-----
                                ; copy n from register passed in to r4
loop          SUBS      r4, r4, #1
                                ; decrement next multiplier
                MULNE    r1, r0, r4
                                ; perform multiply, result in r1
                MOV      r0, r1
                BNE      loop
                                ; go again if not complete
-----
                                ; restore appropriate registers
                                ;     from stack for reentrancy
FactRef       -----
-----
                                ; save appropriate registers
                                ;     on stack for reentrancy
-----
                                ; get parameter by reference into r2
                MOV      r8, r2
                                ; copy n into a temp register
loop1         SUBS      r4, r4, #1
                                ; decrement next multiplier
                MULNE    r1, r2, r4
                                ; perform multiply, result in r1
                MOV      r2, r1
                BNE      loop1
                                ; go again if not complete
-----
                                ; store result (r1) in [r3]
-----
                                ; restore appropriate registers
                                ;     from stack for reentrancy

                END

```

Fill in the blanks above to complete the code.

6. (20 pts) **Exception Handling.** Consider the following reset and undefined handler that was used in your book to define an instruction called ADDSHIFT. The ADDSHIFT instruction added r0 to Rm (a register between r0 and r7 described in the instruction) and left shifts the result by 5. The current behavior can be described as  $r0 = (r0 + Rm) \ll 5$ .

```

; Area Definition and Entry Point
SRAM_BASE      EQU      0x4000                ; start of RAM
Mode_UND       EQU      0x1B
Mode_SVC       EQU      0x13
I_Bit          EQU      0x80
F_Bit          EQU      0x40
                AREA     AddShiftDefined, CODE
                ENTRY

; Exception Vectors
Vectors        LDR      PC, Reset_Addr
                LDR      PC, Undef_Addr
                LDR      PC, SWI_Addr
                LDR      PC, PAbt_Addr
                LDR      PC, DAbt_Addr
                NOP                      ; Reserved Vector
                LDR      PC, IRQ_Addr
FIQHandler     ; do something important
                SUBS     PC, LR, #4        ; return to main program
Reset_Addr     DCD      ResetHandler
Undef_Addr     DCD      UndefHandler
SWI_Addr       DCD      SWIHandler
PAbt_Addr      DCD      PAbtHandler
DAbt_Addr      DCD      DAbtHandler
                DCD      0
IRQ_Addr       DCD      IRQHandler
SWIHandler     B        SWIHandler
PAbtHandler    B        PAbtHandler
DAbtHandler    B        DAbtHandler
IRQHandler     B        IRQHandler
ResetHandler

; Undefined Instruction test
; 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8|7 6 5 4|3 2 1 0
; 0  1  1  1|0  1  1  1|1  1  1  1|0  0  0  0|0  0  0  0|1  1  1  1|1  1  1  1|Rm
; CC = AL    |          OP          | Rn = 0    | Rd = 0    |Not Used    |Rm
; At the beginning of time (after reboot), set up a stack pointer in UNDEF mode
; since we know our simulation will hit an undefined instruction
                MSR      CPSR_c, #Mode_UND:OR:I_Bit:OR:F_Bit
                LDR      SP, =SRAM_BASE+80    ; Initialize stack pointer
                MSR      CPSR_c, #Mode_SVC:OR:I_Bit:OR:F_Bit
                MOV      r0, #124              ; Put test data in r0
                MOV      r4, #0x8B             ; Put test data in r4
ADDSHFTr0r0r4  DCD      0x77F00FF4            ; r0 = (r0 + r4) LSL #5
                MOV      r0, #124              ; Put test data in r0
                MOV      r5, #0x9F             ; Put test data in r5
ADDSHFTr0r0r5  DCD      0x77F00FF5            ; r0 = (r0 + r5) LSL #5
Stop           B        Stop

```

```

UndefHandler
    STMFD    SP!, {r0-r12,LR}      ; Save workspace & LR to stack
    MRS      r0, SPSR              ; Copy SPSR to r0
    STR      r0, [SP, #-4]!        ; Save SPSP to stack
    LDR      r0, [LR, #-4]         ; r0 = 32-bit undefined instruction
    BIC      r2, r0, #0xF00FFFFF   ; clear out all but opcode bits
    TEQ      r2, #0x07F00000       ; r2 = opcode for LargeSub
    BLEQ     ADDSHIFTInstruction   ; if a valid opcode, handle it
    ; otherwise, look for other undefined instructions here...
    LDR      r1, [sp], #4          ; Restore SPSR to r1
    MSR      SPSR_cxsf, r1        ; Restore SPSR
    LDMFD    SP!, {r0-r12,PC}~     ; Return after undefined instructions
ADDSHIFTInstruction
    BIC      r3, r0, #0xFFFFFFF0   ; mask out all bits except Rm
    ADD      r3, r3, #1            ; bump past the SPSR on the stack
    LDR      r0, [sp, #4]          ; grab r0 from the stack
    LDR      r3, [sp, r3, LSL #2]   ; use the Rm field as an offset
    ADD      r0, r0, r3            ; calculate r0+Rm
    MOV      r0, r0, LSL #5        ; r0 = (r0+Rm)<<5
    STR      r0, [sp, #4]          ; store r0 back on the stack
    MOV      pc, lr
    END

```

- (a) (10 pts) Draw the memory map for this program including blocks for the vector table, reset handler, undefined handler, and stack. In addition, describe each word of memory from the start of memory to the beginning of the reset handler.

- (b) (10 pts) Make inline changes to the code to reflect the following new definition of the ADDSHIFT instruction. Bits 19-16 of the instruction will represent the first operand Rn (instead of always being r0). Bits 15-12 of the instruction will represent the destination Rd (instead of always being r0). Finally, r12 will represent the amount of left shifts (instead of always being 5). Thus, the ADDSHIFT instruction would have the following more general behavior  $Rd = (Rn + Rm) \ll r12$ .

7. (21 pts) **Interfacing.** The following code will configure the UART and put one byte into the UART for transmitting. Rewrite the following code (by changing it only where necessary) so that the UART transmission is now 5 bits with even parity and two stop bits. Also, change the stack policy to empty ascending. To receive full credit, do not use EA with the stores and loads, but rather the appropriate increment before (IB), increment after (IA), decrement before (DB), or decrement after (DA).

```

                AREA    UARTDm, CODE, READONLY
PINSEL0        EQU     0xE002C000        ; Controls the function of the pins
UOSTART        EQU     0xE000C000        ; Start of UART0 registers
LCR0           EQU     0xC              ; Line control register for UART0
LSR0           EQU     0x14             ; Line status register for UART0
RAMSTART       EQU     0x40000000        ; Start of onboard RAM for 2104
                ENTRY

start          LDR      sp, =RAMSTART    ; Set up stack pointer
               BL       UARTConfig       ; Initialize/configure UART0
               LDR      r1, =CharData    ; Starting address of characters

Loop           LDRB     r0, [r1], #1      ; Load character, increment address
               CMP      r0, #0           ; Null terminated?
               BLNE     Transmit         ; Send character to UART
               BNE      Loop             ; Continue if not a '0'
Done           B        Done             ; Otherwise, we're done

; Subroutine UARTConfig
; Configures the I/O pins first and sets up the UART control register.
; The parameters are currently set to 8 bits, no parity, and 1 stop bit.
UARTConfig
                STMIB   sp!, {r5, r6, lr}
                LDR      r5, =PINSEL0    ; Base address of register
                LDR      r6, [r5]        ; Get contents
                BIC      r6, r6, #0xF     ; Clear out lower nibble
                ORR      r6, r6, #0x5     ; Sets P0.0 to Tx0 and P0.1 to Rx0
                STR      r6, [r5]        ; Read/Modify/Write back to register
                LDR      r5, =UOSTART
                MOV      r6, #0x83       ; Set to 5 bits, even parity,
                                           ; 2 stop bits, no break tx
                STRB     r6, [r5, #LCR0]  ; Write control byte to LCR
                MOV      r6, #0x61       ; 9600 baud @15MHz VPB clock
                STRB     r6, [r5]        ; Store control byte
                MOV      r6, #3          ; Set DLAB = 0
                STRB     r6, [r5, #LCR0]  ; Tx and Rx buffers set up
                LDMDA    sp!, {r5,r6,pc}

```

```

; Subroutine Transmit
; This routine puts one byte into the UART for transmitting.
Transmit
        STMIB    sp!, {r5, r6, lr}
        LDR      r5, =U0START
wait    LDRB      r6, [r5, #LSR0]      ; Get status of buffer
        CMP      r6, #0x20            ; Buffer empty?
        BEQ      wait                ; Spin until buffer's empty
        STRB      r0, [r5]
        LDMDA     sp!, {r5, r6, pc}

CharData DCB      "Watson. Come quickly!", 0
        END

```

U0LCR	Function	Description
1:0	Word Length Select	00=5-bit, 01=6-bit 10=7-bit, 11=8-bit
2	Stop Bit Select	0=1 stop bit 1=2 stop bits
3	Parity Enable	0=disable parity generation and checking 1=enable parity generation and checking
5:4	Parity Select	00=odd, 01=even, 10=forced "1" stick 11=forced "0" stick
6	Break Control	0=disable break tx 1=enable break tx
7	Divisor Latch Access Bit	0=disable access to divisor latches 1=enable access to divisor latches



Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0, N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-
1111	-	See <i>Condition code 0b1111</i>	-