EE2401 微算機系統 Fall 2019

HW#8 (ARM) (11/30/2019)

Due date: 12/12/2019. Severe penalty will be given to late homework.

Note:

(a) The homework will be graded based on your **answer**. Please read class handouts (ARM-1, ARM-2, ARM-3, and ARM-4)

(b) You are required to **type** your homework (first the problem then your solution) by using a **word processor** and submit in .docx (or .pdf) format under a filename **EE2401f19-hw8-student_no-vn.docx(or .pdf)**, where **student_no** is your student number, e.g., **107061xxx** and **vn** is your version number, e.g., **v3**. You should upload your .doc file in **iLMS** by the specified deadline whenever you have a newer version. Follow the iLMS upload homework process to upload your file.

(c) The homework will be graded based on your **latest version**. Old version(s) will be discarded.

(d) Each homework assignment will have full score of 100 points. 5 points will be deducted if you do not comply with the naming convention. Severe grade penalty will be given to late homework. 20 points will be taken off per day after deadline till zero point. ***Copying is violating the regulations and is definitely not allowed!***

(e) Please treat the above requirements as a kind of training in writing a decent homework report. If you have any problem regarding this homework, **please feel free to consult with TA or teacher**. If you think the time is too short to accomplish this homework, please let me know in class.

1. (20%)

Conditional execution of instruction is a feature of ARM processor. For the C statement: if ((a==b) && (c==d)) e++, please use only three ARM instruction to implement the statement, assuming that variables a, b, c, d, e are stored in r0 ~ r4, respectively.

```
CMP     r0, r1      ; compare a and b
CMPEQ   r2, r3      ; if a == b, then do compare c and d.
ADDEQ   r4, r4, #1  ; if all equal, then e++
```

2. (20%)

Rewrite the following high-level program in ARM assembly language. Assume the variable x and y are passed in r0 and r1, respectively. Try to write an assembly code that uses no more than 7 ARM instructions to implement the function where the last instruction is **mov pc, lr**.

```
int my_func(int x, int y)
{
    if (x >= 0 && y >= 0)
        return (x + y);
    else if (x >= 0 && y < 0)
```

```
        return (x - y);
    else if (x < 0 && y >= 0)
        return (y - x);
    else
        return (x - y);
}
```

I will use r2 as our return register.

We observe that whenever ***(x >= 0 && y < 0) or (x < 0 && y < 0)*** it will return ***(x - y)***

```
CMP     r1,   #0          ; compare y, it will always return x – y if y < 0
SUBLT   r2,   r0,   r1    ; do x – y if y < 0
MOVLT   PC,   lr          ; return if y < 0
CMP     r0,   #0          ; else, compare x
SUBLT   r2,   r1,   r0    ; if x < 0, return y - x
ADDGE   r2,   r1,   r0    ; if x >= 0, return x + y
MOV     PC,   lr          ; return
```

3. (20%)
The SWI instruction is usually used in a user mode program to call for supervisor (OS) service. It contains a 24-bit immediate value (i.e., SWI number) that can pass to the SWI handling routine, e.g., SWI #0x123456. Assume that we want to use **SWI numbers 0-2 only** for performing three different functions (assume their entry address labels are FUN0, FUN1, FUN2), respectively, and each function will take **5 words parameters** passed to SWI handler through **user stack**.
(a) ***Please show how should the user program prepare before invoking SWI*** and
(b) **Write a SWI_handler which can extract the SWI number and the 5-word parameters passed from the SWI instruction and jump to corresponding function.**
The following shows a structure of the part of the user program and SWI handler:

```
User program:
        STMFD       r13!,….                     ;push 5 parameters to the user
stack
        …(a)                    ;
        SWI         SWI_number  ;invoke SWI with SWI_number
        …

SWI_handler
        STMDB       r13!, {r0-r12, lr}          ;push to svc stack used reg. & ret
addr.
        …(b)
        …
        LDMIA       r13!, {r0-r12, pc}^     ;SWI return
FUNCTABLE
        DCD   FUN0
        DCD   FUN1
        DCD   FUN2
```

(a)
User Program:

We assume that the five parameters were previously stored in R0, R1, R2, R3, R4. Hence, we first the parameters to the stack. Then, since the R13 will be changed if we switch to SVC stack, so after the parameters pushed to the stack, we have to store the stack pointer to R1, in order to preserve the location of the five parameters.

(b)
SWI_handler:

First we push the contents in the registers and SPSR to the SVC stack for later return. Then, we pop out the parameters to register R6-R10, by using R1, since it is previously pointed to the parameters. Then, we extract the SWI number and we mask the unwanted bits of the SWI number. Then we call the SWI, which will call the FUNCTABLE. Finally, pop out all the values and return.

User Program:

```
        STMFD   r13! {r0-r4}          ; push 5 parameters to the user stack
        MOV     r1, r13               ; R1 now is the pointer of user stack
        SWI     SWI_number            ; invoke SWI with SWI_number
SWI_handler:
        STMDB   r13!, {r0-r12, lr}    ;push to svc stack used reg. & ret
        MRS     r0,     spsr          ; get SPSR
        STMDB   r13!,   {r0}          ; and push it
        LDMIA   r1!,    {r6-r10}      ; pop 5 needed parameters to r6-r10
        LDR     r0,     [lr, #-4]     ; extract SWI number to R0
        BIC     r0,     r0, #0xffffff0 ; in r0, since we only need the last byte
        BL      ISR_SWI               ; pass it to the ISR
        LDMIA   r13!,   {r0}          ; pop SPSR
        MSR     spsr,   r0;
        LDMIA   r13!,   {r0-r12, pc}^ ; SWI return

ISR_SWI                               ; this ISR is a jump table subroutine
        ADR     r1, FUNCTABLE         ; get the label address for FUNCTABLE
        LDRLS   pc, [r1, r0, LSL #2]  ; pc = r1 + r0 * 4

FUNCTABLE
        DCD   FUN0
        DCD   FUN1
        DCD   FUN2
```

4. (20%)

Consider a memory intensive task of copying a zero-terminated string of characters from in to out. It converts the string to lowercase in the process. The following is the C program for doing this task:

```
void  str_2lower(char *out, char *in)
{
        unsigned int    c;
        do
        {
                c = *(in++);
```

```
            if (c>='A'&& c<='Z')
            {
                    c = c + ('a'–'A');
            }
            *(out++) = (char)c;
        } while (c);
}
```

Below is a possible instruction sequence of an ARM assembly subroutine _str_2lower of this C function. Assume that the out and inpointers are passed to _str_2lower in r0 and r1, respectively. No registers (except possibly pc and lr) should be altered by the subroutine. Please fill the missing parts in the blanks.
Note: ASCII code 'A' = 0x41, 'B' = 0x42, ..., 'Z'=0x5A, 'a' = 0x61, 'b' = 0x62,...

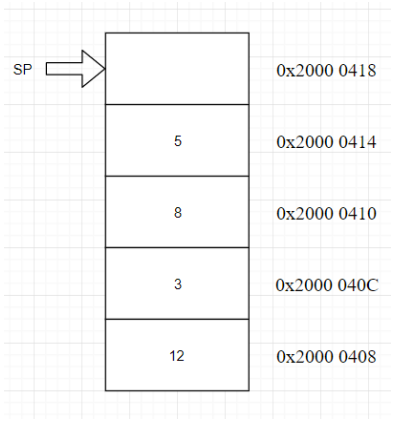
```
_str_2lower
        __1___          r13!,{r0-r3}        ; push r0~r3
        __2___          r2,[r1],#1          ; c = *(in++)
        SUB             r3,r2,#0x41         ; r3 = c –'A'
        CMP             r3,#0x19            ; if (c <= 'Z'-'A')
        __3___          r2,r2,#0x20         ; c += 'a'-'A'  convert to
lowercase
        __4___          r2,[r0],#1          ; *(out++) = (char) c
        CMP             r2,#0               ; if (c!=0)
        BNE             _str_2lower         ; goto _str_2lower
        __5___          r13!,{r0-r3}        ; pop r0~r3
        MOV             pc, r14             ; else return
```

```
1.      STMFD           ; push ro-r3 to the stack with (FD) type
2.      LDRB            ; load byte into a register
3.      ADDLS           ; if (c <= 'Z' – 'a'), then c += 'a' – 'A'
4.      STRB            ; store byte into out
5.      LDMFD           ; pop using (FD)
```

5. (20%)
Assume the stack pointer (SP) is initialized to 0x2000 0408. Registers R0, R1, R2 and R12 are initialized to 12, 3, 8 and 5 respectively. Answer the following:
(a) Show **the content of the stack** and the **SP** after the following sequence of operations:

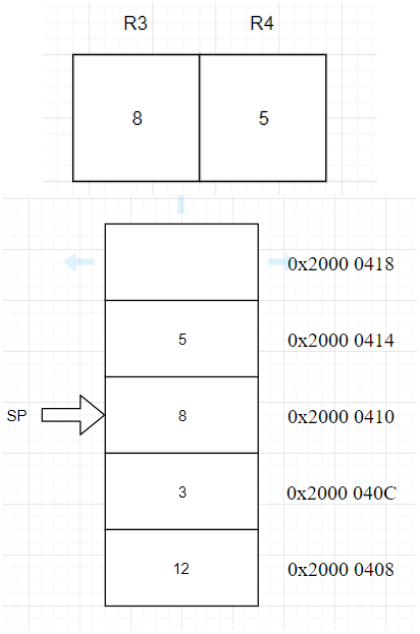
```
        STMEA SP!, {R0-R2,R12}
```

The content in the registers will be stored into the stack, in the order of low to high. Also, we are using empty ascending(EA), it will point to an empty, highest stack address.



(b) Given the state of the stack after part a), show the **content of the stack**, the **SP** and **registers R3, R4** after the following operation: LDMEA SP!, {R3-R4}

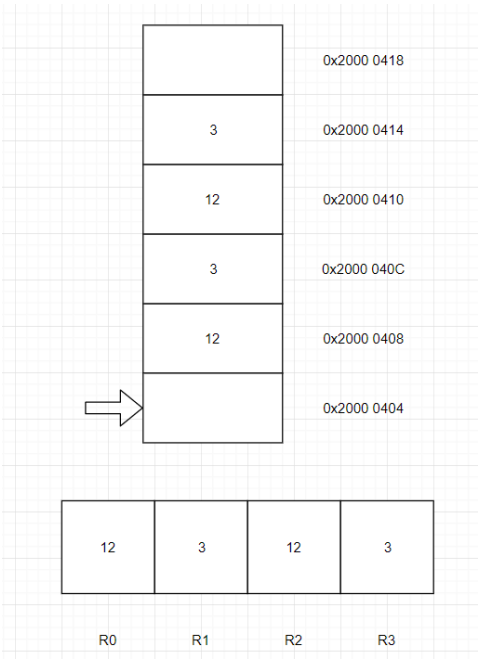
The highest two data will be loaded to R3, R4.



(c) Given the state of the stack after part b), show the **content of the stack**, the **SP** and **registers R0- R3** after the following operations:

STMEA SP!, {R0-R1}
LDMEA SP!, {R0-R3}

First, the content of R0, R1, which is 12 and 3, will be stored to the stack, SP pointing to 0x2000 0418. Then all the value will be popped to the register. Since we are using the EA mode, the stack must point to an empty address, which is 0x2000 0404.


0x2000 0418
3
0x2000 0414
12
0x2000 0410
3
0x2000 040C
12
0x2000 0408
0x2000 0404
12
3
12
3
R0
R1
R2
R3