Practice: More on Assembly Programming

Call and return sequence

1. Stack Frames

- The block of information stored on the stack to implement a subroutine call and return

  - A subroutine foo gets two arguments in R0 and R1 from the caller and returns the result in R0
    - foo(a, b) = bar(a^b)
    - foo calls bar
  - A subroutine bar gets one argument in R0 from the caller and returns the result in R0
    - bar(a) = inc(a × a)
    - bar calls inc
  - A subroutine inc gets one argument in R0 from the caller and returns the result in R0
    - inc(a) = a + 1

1. Stack Frames (contd.)

```
@=====================
@ inc(x)
@=====================
inc:  ADD  R0, R0, #1  @ Increment R0, R0 contains the
                       @   return value
      MOV  PC, LR      @ Return to the caller
```

1. Stack Frames (contd.)

```
@====================
@ bar(x)
@====================
bar:  SUB  SP, SP, #4  @ Decrement SP by 4
      STR  LR, [SP]    @ Save LR on the stack
@====================
      MUL  R1, R0, R0
      MOV  R0, R1      @ The argument for inc is in R0
      BL   inc         @ Call inc. When returning from
                       @   inc, R0 contains the result
@====================
      LDR  LR, [SP]    @ Restore LR from the stack
      ADD  SP, SP, #4  @ Increment SP by 4
      MOV  PC, LR      @ Return to the caller
```

1. Stack Frames (contd.)

```
@====================
@ foo(x, y)
@====================
foo:  SUB  SP, SP, #4  @ Decrement SP by 4
      STR  R5, [SP]    @ Save R5 on the stack
      SUB  SP, SP, #4  @ Decrement SP by 4
      STR  R6, [SP]    @ Save R6 on the stack
      SUB  SP, SP, #4  @ Decrement SP by 4
      STR  LR, [SP]    @ Save LR on the stack
@====================
      MOV  R5, #1
L1:   CMP  R1, #0
      BEQ  DONE
      MUL  R6, R5, R0
      MOV  R5, R6
      SUB  R1, R1, #1
      B    L1
DONE: MOV  R0, R5      @ The argument for bar is in R0
      BL   bar         @ Call bar. When returning from
                       @   bar, R0 contains the result
@====================
      LDR  LR, [SP]    @ Restore LR from the stack
      ADD  SP, SP, #4  @ Increment SP by 4
      LDR  R6, [SP]    @ Restore R6 from the stack
      ADD  SP, SP, #4  @ Increment SP by 4
      LDR  R5, [SP]    @ Restore R5from the stack
      ADD  SP, SP, #4  @ Increment SP by 4
      MOV  PC, LR      @ Return to the caller
```

2. Printing a Character or an Integer

- printInt
  - Before you call printInt, you need to set up r0 and r1.
    - r0 - the screen address map to which the integer is printed
    - r1 - the integer
- On return, r0 contains (addr + the number of characters printed).
  - The next address on the screen for printing

- printCh
  - Before you call printCh, you need to set up r0 and r1
    - r0 - the screen map to which the character is printed
    - r1 - the character in ascii
  - On return, r0 contains (addr + 1)

2. Printing a Character or an Integer (contd.)

- Subroutine **`asciiStr2Int`**
  - Converts a null-terminated string to an integer
    - "3456" → 3456
  - When called, r0 contains the address of the null-terminated string
  - On return r0 contains the converted integer

2. Printing a Character or an Integer (contd.)

```
@====================
@ asciiStr2Int
@====================
asciiStr2Int:
      ldr  r12, r0          @ R12 contains the address
                            @   of the string
      eor  r0, r0, r0       @ Set r0 to 0
      mov  r10, #10         @ Set r10 to 10
      ldrb r11, [r12]       @ Load the 1st digit to r11
LOOP:
      cmp  r11, #0          @ Does r11 contains '\0' (NULL)?
      beq  DONE             @ If so, done
      mul  r1, r0, r10      @
      sub  r11, r11, #0x30  @ Get the value of the digit,
                            @   '0' is 0x30 in ASCII
      add  r0, r1, r11      @
      add  r12, r12, #1     @
      ldrb r11, [r12]       @ Get the next digit
      b    LOOP             @
DONE:
      mov  pc, lr           @
```

2. Printing a Character or an Integer (contd.)

```
@====================
@ main
@====================
main:
      ldr  r0, addr_str     @ R0 contains the address
                            @   of the string
      bl   asciiStr2Int     @ Call asciiStr2Int
      mov  r1, r0           @ R1 contains the integer
      ldr  r0, addr_screen  @ R0 contains the location
                            @   on the screen
      bl   printInt         @ Call printInt
      b    halt             @
addr_str
      .word 0x9000
addr_screen
      .word 0xD0000
```

3. A Simple Calculator

- The null-terminated input string that contains an arithmetic expression in the RPN is stored in a memory location whose address is stored in the location labeled “input_str_addr”
- The numbers and operators are delimited by a space character (0x20 in ASCII)
  - A buffer is used to store a string that represents a number
  - This will be converted to an integer
- We want to know the value of the input expression
  - Use a user defined stack (not a call stack)
    - An empty descending stack
    - The stack bottom is located at “stack_bottom_addr”

3. A Simple Calculator (contd.)

```
@====================
@ Simple Calculator
@====================
    ldr  r5, stack_bottom_addr   @ R5 is a pointer to the top of the user stack
    ldr  r7, input_str_addr      @ R7 contains the address of the input string
    sub  r7, r7, #1
LOOP1:
    add  r7, r7, #1              @ Increment r7 by one
    ldrb r6, [r7]                @ Load the next char in r6
LOOP1_NUMBER:
    cmp  r6, #0                  @ Check if the char is '\0' (NULL)?
    beq  ALL_DONE                @ If so, done!
    cmp  r6, #0x2b               @ Check if the char is '+'
    beq  ADD                     @ If so, perform addition
    cmp  r6, #0x2d               @ Check if the char is '-'
    beq  SUBTRACT                @ If so, perform subtraction
    cmp  r6, #0x20               @ Check if the char is a space (' ')
    beq  LOOP1                   @ If so, skip to the next char
```

3. A Simple Calculator (contd.)

```
NUMBER:                                 @ Otherwise, it is the first char of a number
    ldr  r8, =buffer                    @ R8 contains the address of the buffer
LOOP2:
    strb r6, [r8]                       @ Put the char in to the buffer
    add  r8, r8, #1                     @ Increment the buffer pointer
    add  r7, r7, #1                     @ The address of the next char
    ldrb r6, [r7]                       @ Load the next char in r6
    cmp  r6, #0x20                      @ Check if the char is a space (' ')
    beq  CONVERT_STR_TO_INT:            @ If so, a complete string for a
                                        @   number is recognized
    cmp  r6, #0                         @ Check if the char is '\0'
    bne  LOOP2                          @ If not, we are in the middle of a number
CONVERT_STR_TO_INT:                     @ Convert the string to an integer
    mov  r9, #0
    strb r9, [r8]                       @ Put '\0' at the end of the buffer
    ldr  r0, =buffer                    @ The first argument of asciiStr2Int
    bl   asciiStr2Int                   @ Call asciiStr2Int
    sub  r5, r5, #4                     @ Decrement the user stack pointer by four
    str  r0, [r5]                       @ Push the integer onto the user stack
    b    LOOP1_NUMBER
```

3. A Simple Calculator (contd.)

```
ADD:
    ldr   r4, [r5]              @ Pop the second operand of +
    add   r5, r5, #4            @ Increment the user stack pointer by four
    ldr   r3, [r5]              @ Pop the first operand of +
    add   r3, r3, r4            @ Perform addition
    str   r3, [r5]              @ Push the result on to the user stack
    b     LOOP1
SUBTRACT:
    ldr   r4, [r5]              @ Pop the second operand of -
    add   r5, r5, #4            @ Increment the user stack pointer by four
    ldr   r3, [r5]              @ Pop the first operand of -
    sub   r3, r3, r4            @ Perform subtraction
    str   r3, [r5]              @ Push the result on to the user stack
    b     LOOP1
ALL_DONE:
    ldr   r0, screen_addr       @ The first argument of printInt
    ldr   r1, [r5]              @ The second argument of printInt
    bl    printInt              @ Call printInt
    b     halt
```

3. A Simple Calculator (contd.)

```
screen_addr:
     .word 0xD0000
input_str_addr:
     .word 0x9000
stack_bottom_addr:
     .word 0xa000
buffer:
     .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

4. Recursion vs. Iteration

- Iterative subroutine
  - Repetition of a sequence of instructions (using a loop)
- Recursive subroutine
  - The subroutine calls itself
- Any recursive function
  - Iteration + stack

4. Recursion vs. Iteration (contd.)

- Factorial function
  - An recursive definition

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times fact(n-1) & \text{if } n > 0 \end{cases}$$

```
@============================
@ Fact (iterative version)
@============================
fact:
    mov   r1, #1
LOOP:
    cmp   r0, #0          @ Check if r0 is 0
    beq   DONE            @ If so, done
    mul   r1, r1, r0
    sub   r0, r0, #1      @ Decrement r0 by one
    b     LOOP
DONE:
    mov   r0, r1          @ The return value is in r0
    mov   pc, lr          @ Return to the caller
```

```
//============================
// Fact (iterative version)
//============================
int fact(int n)
{
    int m = 1, i;
    for (i = 1; i <= n; i++)
        m = m * i;

    return m;
}
```

4. Recursion vs. Iteration (contd.)

```
@============================
@ Fact (recursive version)
@============================
fact:
    sub   sp, sp, #4
    str   r4, [sp]
    sub   sp, sp, #4
    str   lr, [sp]
@============================
    cmp   r0, #0          @ Check if r0 (n) is 0
    beq   BASE            @ If so, done
    mov   r4, r0          @ R4 contains n
    sub   r0, r0, #1      @ R0 contains n-1
    bl    fact            @ Call fact(n-1)
    mul   r0, r4, r0      @ n × fact(n-1)
    b     DONE
BASE:
    mov   r0, #1          @ The return value 1 is in r0
@============================
DONE:
    ldr   lr, [sp]        @ Restore lr from the stack
    add   sp, sp, #4
    ldr   r4, [sp]        @ Restore lr from the stack
    add   sp, sp, #4
    mov   pc, lr          @ Return to the caller
```

```
//============================
// Fact (recursive version)
//============================
int fact(int n)
{
    if (n == 0)
        return 1;
    else
        return n*fact(n-1);
}
```

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times fact(n-1) & \text{if } n > 0 \end{cases}$$

A subroutine (function) is recursive if it calls itself either directly or indirectly

4. Recursion vs. Iteration (contd.)

- Fibonacci numbers are defined by

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
- Can you write an assembly subroutine and a C function that takes an integer n as an argument and returns the n^{th} Fibonacci number?
  - Both iterative version and recursive version

4. Recursion vs. Iteration (contd.)

```
mov r0, #7
bl fibo
b halt

@@@@@@@@@@@@@
@ Recursion @
@@@@@@@@@@@@@

fibo:
@=======================
    sub sp, sp, #4
    str r4, [sp]
    sub sp, sp, #4
    str r5, [sp]
    sub sp, sp, #4
    str lr, [sp]
@=======================
    cmp r0, #1
    beq DONE
    cmp r0, #0
    beq DONE
    mov r4, r0
    sub r0, r4, #1
    bl fibo
    mov r5, r0
    sub r0, r4, #2
    bl fibo
    add r0, r5, r0
    b DONE

DONE:
@=======================
    ldr lr, [sp]
    add sp, sp, #4
    ldr r5, [sp]
    add sp, sp, #4
    ldr r4, [sp]
    add sp, sp, #4
    mov pc, lr
@=======================

addr_screen: .word 0xd0000
```

Exercise

- Try to run all the example code
  - Stack frames – foo (including bar and inc)
  - Printing a character or an Integer – asciistr2Int
  - A Simple calculator – simpleCalc
  - Recursion vs. Iteration – factorial
  - Recursion vs. Iteration – fibonacci