Lab2 ARM Assembly II

實驗二 ARM Assembly II

1. Lab objectives 實驗目的

- Familiar with basic ARMv7 assembly language.
- In this Lab, we will learn the topics below.
  - How to do multi-word arithmetic.
  - The alignment issue of the arm assembly
  - How to access stack memory.
  - How to implement a recursive call.
- 熟悉基本 ARMv7 組合語言語法使用。
- 在這次實驗中需要同學了解
  - 如何執行多字元算術
  - Arm 組合語言的對齊問題
  - 如何存取記憶體堆疊
  - 如何實現遞迴呼叫

2. Lab principle 實驗原理

Please refer to the course ARMASM lecture slide and STM32L4x6 Reference manual.
請參考上課 ARMASM 講義與 STM32L4x6 Reference manual。

3. Steps 實驗步驟

3.1. Karatsuba algorithm

The Karatsuba algorithm is a fast multiplication algorithm using the divide and conquer trick. By the Karatsuba algorithm, we get the following equation, where X and Y are two n-bits integers; X_L, X_R are the leftmost, and rightmost halves of X, and Y_L, Y_R is the leftmost and rightmost halves of Y.
Karatsuba算法是一種使用分而治之技巧的快速乘法演算法。通過 Karatsuba 演算法，我們得到以下等式。其中 X 和 Y 是兩個 n 位元整數; X_L, X_R 是 X 的最左，最右一半，而 Y_L, Y_R 是 Y 的最左，最右一半。

$$X*Y = 2^n * X_L Y_L + 2^{n/2} * [(X_L + X_R)(Y_L + Y_R) - (X_L Y_L + X_R X_R)] + X_R Y_R$$

It can also be simply prove by equations, (1) and (2)
這可以簡單由等式 (1), (2) 證明

$$X*Y = (2^{n/2} * X_L + X_R)(2^{n/2} * Y_L + Y_R)$$
$$= 2^n * X_L Y_L + 2^{n/2} *(\mathbf{X_L Y_R + Y_L X_R}) + X_R Y_R \text{ --- (1)}$$

$$(X_L + X_R)(Y_L + Y_R) = X_L Y_L + \mathbf{X_L Y_R + X_R Y_L} + X_R Y_R$$
$$\Rightarrow (X_L + X_R)(Y_L + Y_R) - (X_L Y_L + X_R Y_R) = \mathbf{X_L Y_R + X_R Y_L} \text{ --- (2)}$$

The trick part is that it replaces $\mathbf{X_L Y_R + X_R Y_L}$ in (1) by equation (2) and thus, we need only three n/2-bits multiplications compared to equation (1) which four multiplications are required.
技巧所在是使用等式 (2) 代替 (1) 中的 $\mathbf{X_L Y_R + X_R Y_L}$，由此，我們只需要三個 n/2-bits 乘法。相比於等式 (1)卻需要四次。

Requirement: Please implement the Karatsuba algorithm which accepts two 32-bit unsigned integers "X, Y", and stores the result of X times Y into the variable “result”.
請實現 Karatsuba **算法，該算法接受兩個**32**位元無號整數** "X, Y"，**並將** X **乘以** Y **的結果存儲到變量** "result" **中。**

(Hint: The output can be a 64-bits integer. You may need these instructions, ADC, STRD.
提示：輸出可能為 64 **位元整數。 您可會用到** ADC, STRD **這些指令。**)

```
   .syntax unified
   .cpu cortex-m4
   .thumb

.data
   result: .zero 8

.text
  .global main
  .equ X, 0x12345678
  .equ Y, 0xABCDEF00

main:
  LDR R0, =X
  LDR R1, =Y
  LDR R2, =result
  BL kara_mul

L:
  B L

kara_mul:
    //TODO: Separate the leftmost and rightmost halves into
different registers; then do the Karatsuba algorithm.
    BX LR
```

3.2. Parentheses Balance

A well-known application of the stack is to check the balance of parentheses. We read the elements of the infix expression from left to right. When we meet an opening parenthesis, we push it onto the stack, and when we meet a closing parenthesis, we pop the element from the stack to match it. Finally, we check if the stack is empty.
堆疊的一個著名應用是檢查括號的平衡。我們從左到右讀取中序表達式的元素。當我們遇到開括號時，我們將其推入堆疊，當我們遇到閉括號時則從堆疊中彈出元素與之匹配。最後我們檢查堆疊是否為空。

Reference:
https://people.cs.clemson.edu/~goddard/texts/dataStructCPP/chap10.pdf

Requirement: Please modify the code provided below, and implement a subroutine "pare_check" which accepts a pointer("R0") pointing to a "***infix_expr"***. It will check the parentheses balance. If there was an error occurred it would set "R0" to -1. Otherwise, "R0" will be set to 0.
(Note: You must allocate space in the data section and set that space as your stack, and use PUSH, POP instructions to access your stack memory.

請修改下面提供的程式碼，並實做子程序“ pare_check”，該子程序接受指向“ infix_expr”的指針（“ R0”）。它將檢查括號內的匹配。如果發生錯誤，則將“ R0”設置為 -1。否則，“ R0”將被設置為 0。
注意：您必須在 data section 中分配空間並將該空間設置為堆疊，並使用 PUSH，POP 指令存取記憶體堆疊。

infix_expr: "infix_expr" is an infix expression with the following constraints. The string of the infix expression is ended with the character '\0' and only the following elements are accepted.

"infix_expr"是具有以下限制的中序表達式。中序表達式的字符串以字符 '\0' 結尾，並且只接受以下元素。

Symbol	' '	*	+	-	/	[	]	{	}	0~9
ASCII	32	42	43	45	47	91	93	123	125	48~57

(Hint: You can use MSR to modify the value of MSP(Main Stack Pointer)
可以利用 MSR 來修改 MSP(Main Stack Pointer) 的值)

```
   .syntax unified
   .cpu cortex-m4
   .thumb
.data
   infix_expr:  .asciz  "{-99+ [ 10 + 20-0] }"
   user_stack_bottom: .zero 128

.text
  .global main
  //move infix_expr here. Please refer to the question below.

main:
  BL stack_init
  LDR R0, =infix_expr
  BL pare_check
L:  B L

stack_init:
    //TODO: Setup the stack pointer(sp) to user_stack.
    BX LR

pare_check:
    //TODO: check parentheses balance, and set the error code
            to R0.
    BX LR
```

3.3. Stein's GCD Algorithm

Stein's GCD algorithm is a binary GCD algorithm that uses arithmetic shift, comparison, and subtraction instead of division, and it has been proven to be more efficient than the Euclidean algorithm. Here, we show the implementation of the C language. You can check the link below for more information.
Stein的GCD演算法為二進制GCD演算法，使用算術移位，比較和減法取代除法，並且被證明相較於歐幾里得演算法有更高的效率。在這裡，我們展示了C語言的實現。您可以檢查下面的鏈接以獲取更多信息。

Reference: https://en.wikipedia.org/wiki/Binary_GCD_algorithm#Algorithm

```
/* Stein's Algorithm (C version)*/
int GCD(int a, int b) {
   if (a == 0)  return b;    if (b == 0)  return a;
   if (a % 2 == 0 && b % 2 == 0)    return 2 * GCD(a >> 1, b >> 1);
   else if (a % 2 == 0)   return GCD(a >> 1, b);
   else if (b % 2 == 0)   return GCD(a, b >> 1);
   else   return GCD(abs(a - b), min(a, b));
}
```

Requirement: Please implement Stein's GCD algorithm which accepts two arguments "m, n", and stores the GCD(Greatest Common Divisor) of them into the variable “result”. Please implement the algorithm (Stein’s Algorithm) in recursive mode.

請實現Stein的GCD算法，該算法接受兩個引數 "m, n",並將它們的GCD（最大公約數）存儲到變數 "result" 中。 請以遞歸迴方式實做該演算法。

```
.data
  result: .word  0
  max_size: .word  0

.text
  m: .word  0x5E
  n: .word  0x60

GCD:
  //TODO: Implement your GCD function
  BX LR

.global main
main:
  // r0 = m, r1 = n
  BL GCD
  // get return val and store into result
```

3.4. Question 實驗課問題

Question 1: What is "caller-save register"? What is "callee-save register"? What are their pros and cons?
甚麼是 caller-save register? 甚麼是 "callee-save register"? 各有甚麼優缺點？

Question 3: When recursive functions are executing the self-calling. Which registers should be backed up to stack?
當遞迴函數在執行自調用時。哪些暫存器應該要被備份到記憶體堆疊？

Question4: If we want to use STM, LDM instructions to replace POP, PUSH instructions. Which suffix should be added?
如果我們想用 STM, LDM 指令來取代 POP, PUSH 指令。分別該加上哪種後綴？

3.5. Reference & Hint 參考資料與提示

Hint 1: Given a 32-bits integer. How can we get the leftmost and rightmost halves by logical operations such as AND, OR, NOT, XOR, SHIFT ...etc?

給定一個32位元整數。我們如何通過邏輯運算（例如 AND, OR, NOT, XOR, SHIFT ....等）獲得最左邊和最右邊的一半位元？

Hint 2: In the entire memory space, where can be used as our memory stack?
在整個記憶體區段中，哪裡可以作為我們的記憶體堆疊使用？

Hint 3: If we redefine the variable "infix_expr" in the text section, the builder will raise an error. Why? And one of the brutal force methods to fix this bug is adding 1, 3, 5 ... spaces in the string. Is there any other way?

如果我們在文本部分中重新定義變量“ infix_expr”，則構建器將引發錯誤。 為什麼？而解決該錯誤的一種暴力法是在字串中加入1,3,5 ...個空格。 還有其他辦法嗎？

Hint 4: To implement Stein's GCD algorithm, can we actually avoid the use of any multiplication, division and modulo instructions? why?

實做 Stein 的 GCD 演算法時，我們真的可以避免使用任何乘法，除法和取餘指令嗎？ 為什麼？