Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Department of Computer Science

Computer Architecture II
CSU34021

Tutorial 2: Solutions
Intel's 64-bit Assembly with C/C++

Syed Asad Alam

Document History

Rev.	Date	Comment	Author
1.0	28-11-2020	Tutorial 2 Solution released	SAA

1 Learning Outcomes

This lab satisfies the following learning outcomes of the course:

LO1 Write simple x64 assembly language functions

LO2 Explain the x64 procedure calling conventions

LO3 Write programs that mix C/C++ and x64 assembly language functions

2 Exercises

2.1 Program 1

The following procedure calculates a Fibonacci number by recursion:

```
long long fibonacci_recursion(long long fin)
{
  if(fin <= 0)
    return fin;
  else if (fin == 1)
    return 1;
  else
    return fibonacci_recursion(fin-1) + fibonacci_recursion(fin-2);
}
```

Assembly Code

The corresponding assembly code is:

```
public fibX64

fibX64: ;; This is a non-leaf function so need to align the stack
        ;; preserve the argument in the shadow space
        mov [rsp+32], rcx

        ;; if argument <= 0, return 0
        cmp rcx, 0
        mov rax, rcx
        jle ret_f

        ;; if argument == 1, return 1
        cmp rcx, 1
        mov rax, rcx
        je ret_f

        ;; if not, call fibonacci again, twice
        dec rcx
        sub rsp, 32       ;; allocate shadow space
        call fibX64
        add rsp, 32       ;; remove shadow space
        mov [rsp+16], rax ;; saving the first return argument in the shadow space

        mov rcx, [rsp+32] ;; retreiving the argument again
        sub rcx, 2        ;; in order to call fib(n-2)
        sub rsp, 32       ;; allocate more shadow space
        call fibX64
```

```
        add rsp, 32         ;; remove shadow space

        ;; adding two return values together
        add rax, [rsp+16]
ret_f:  ret
```

2.2 Program 2

The following procedure takes a user input through scanf, calculates the sum of the input arguments and user input and prints the result while returning the sum. The user input should also be accessible from other C/C++ functions:

```
long long use_scanf(long long a, long long b, long long c)
{
  long long sum = a+b+c;
  long long inp_int;

  printf("Please enter an integer: ");
  scanf("%lld", &inp_int);

  sum = sum+inp_int;

  printf("The sum of proc. and user inputs (%lld, %lld, %lld, %lld): %lld\n",
         a,b,c,inp_int,sum);

  return sum;
}
```

The scanf function requires two arguments. The first one is the format specifier (%lld) which can be defined as a string, similar to the string needed for printf and address of this string loaded as an argument and the second argument is the address of variable in memory where it will return the user input (as shown in the "C" code by &inp_int)

Assembly Code

The corresponding assembly code is:

```
public use_scanf

use_scanf:

        xor rax, rax            ;; clearing eax
        lea rax, [rcx+rdx]      ;; adding the first two arguments
        add raX, r8             ;; adding the third argument
        mov [rsp+32], rax       ;; preserving the sum in the shadow space

        ;; preserving the inputs for later user
        mov [rsp+24], rcx
        mov [rsp+16], rdx
        mov [rsp+8], r8

        ;; preparing for calling scanf (including printf)
        ;; first prompt
        sub rsp, 32             ;; shadow space
        lea rcx, inp_str        ;; address of string for prompting user
        call printf             ;; calling printf to display user prompt
        ;; then user input
```

```
        lea rcx, inp_fmt         ;; format of input
        lea rdx, inp_int         ;; address of the place holder
        call scanf               ;; calling the scanf function
        add rsp,32               ;; de-allocating the shadow space

        ;; Retreiving the sum from stack
        mov rax, [rsp+32]

        ;; Adding user input, address of which is in rdx
        mov rbx, inp_int         ;; retreiving the user input, also need later on
        add rax, rbx

        ;; Printing the final sum
        mov [rsp+32], rax        ;; preserving the sum for return
        sub rsp, 48              ;; shadow space for 6 arguments
        lea rcx, out_str         ;; address of string to print
        mov rdx, [rsp+72]        ;; first arg to this proc
        mov r8, [rsp+64]         ;; second arg to this proc
        mov r9, [rsp+56]         ;; third arg to this proc
        mov [rsp+32], rbx        ;; fourth arg to this proc on stack
        mov [rsp+40], rax        ;; fifth arg to this proc on stack
        call printf
        add rsp, 48

        mov rax, [rsp+32]        ;; retreiving the final sum for returning
        ret
```

2.3 Program 3

The following are two procedures, with max5 calling max to calculate its return value.

```
_int64 max(_int64 a, _int64 b, _int64 c) {
    _int64 v = a;
    if (b > v)
        v = b;
    if (c > v)
        v = c;
    return v;
}

// inp_int: The user input in Program '1'
_int64 max5(_int64 i, _int64 j, _int64 k, _int64 l)
{
  return max(max(inp_int, i, j), k, l);
}
```

Assembly Code

The corresponding assembly code is:

```
;; max5 with max

;; the max function
;; three arguments (a,b,c), all 64-bits
;; rcx : a
;; rdx : b
```

```
;; r8  : c
max:    mov rax, rcx      ; v = a
        cmp rdx, rax      ; if (b>v)
        jle max0
        mov rax, rdx      ; v = b
max0:   cmp r8, rax       ; if (c>v)
        jle min1
        mov rax, r8       ; v = c
min1:   ret               ; return v


;; max5 takes in '4' arguments (i,j,k,l), all 64-bit integers, and calls max twice
;; first max call is: t = max(inp_int,i,j), where inp_int is the global variable whose v
;; second max call max(t,k,l);
;; rcx : i
;; rdx : j
;; r8  : k
;; r9  : l
public max5

max5:   ;; preserving the last two arguments in the shadow space
        ;; allocated by main for max5
        mov [rsp+32], r9
        mov [rsp+24], r8

        ;; preparing the arguments for 1st call to max
        sub rsp,32                ; shadow space for max
        mov r8,  rdx              ; third argument, 'j'
        mov rdx, rcx              ; second argument, 'i'
        mov rcx, inp_int          ; first argument, the global variable in rcx
        call max                  ; max(inp_int,i,j)
        ;; preparing the arguments for 2nd call to max
        mov rcx, rax              ; the first argument is the return value of the previous
        mov rdx, [rsp+56]         ; second argument, 'k'
        mov r8, [rsp+64]          ; third argument, 'l'
        call max                  ; max(max(inp_int,i,j),k,l)
        add rsp, 32               ; deallocating shadow space
        ret
```

2.4 Stack Diagram

The stack diagram shows more details than required originally. The stack will reach the maximum depth twice as the recursive Fibonacci function is called twice for two arguments as it reaches the base case. The stack diagram in Fig. 1 shows how the stack is populated as it traverses the recursive function calls for different arguments.

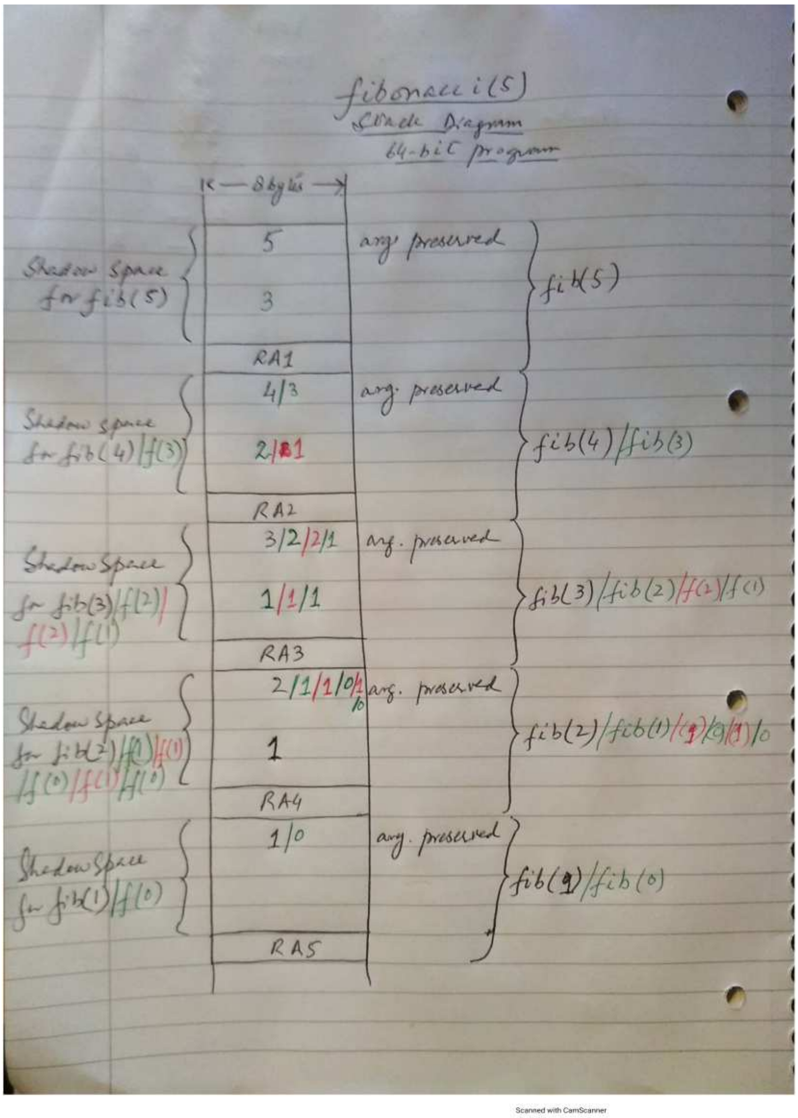

Figure 1: Stack diagram for recursive Fibonacci function.