Xi'an Jiaotong-Liverpool University

西交利物浦大學

Functions in ARM

Jianjun Chen

Contents

- Implementing ARM Functions
- Case study: Recursive Functions and Optimisation

Functions and Stacks

Implementing functions that support multi-level calls

Four stack types

Function Calls in ARM

- Elements of a function: function name, return value, parameters, local variables, function logic.

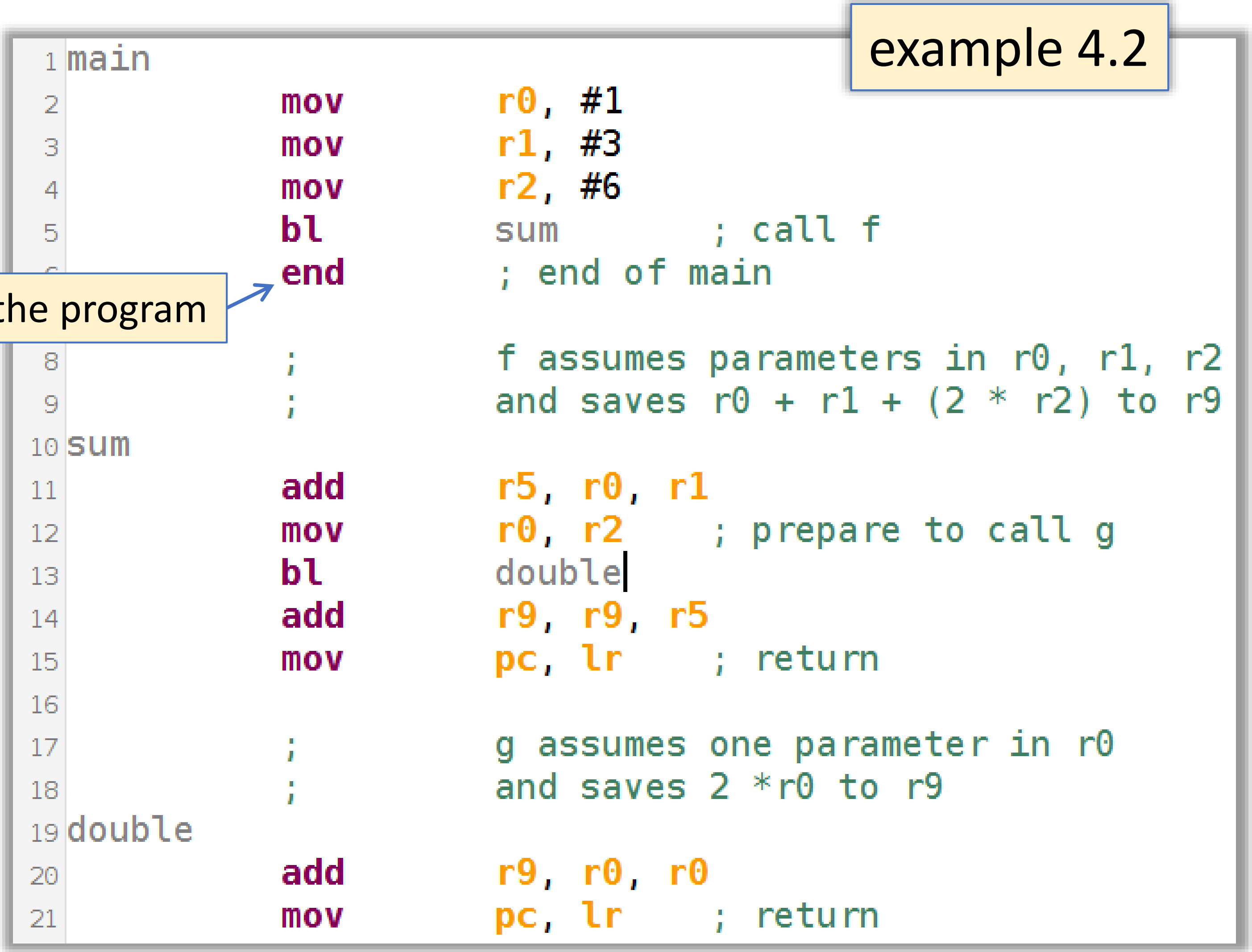
- ARM's branch and link instruction, `BL`, automatically saves the return address in the register R14 (i.e. LR).

- We can use “`MOV PC, LR`” at the end of the subroutine to return back to the instruction after the subroutine call `BL SUBROUTINE_NAME`.
  - A `SUBROUTINE_NAME` is a label in the ARM program.

- However, BL and BX/MOV alone are not enough to support function calls:

Any Problems?

example 4.2



```
main
            mov         r0, #1
            mov         r1, #3
            mov         r2, #6
            bl          sum         ; call f
            end         ; end of main

            ;           f assumes parameters in r0, r1, r2
            ;           and saves r0 + r1 + (2 * r2) to r9
sum
            add         r5, r0, r1
            mov         r0, r2      ; prepare to call g
            bl          double
            add         r9, r9, r5
            mov         pc, lr      ; return

            ;           g assumes one parameter in r0
            ;           and saves 2 *r0 to r9
double
            add         r9, r0, r0
            mov         pc, lr      ; return
```

end stops the program

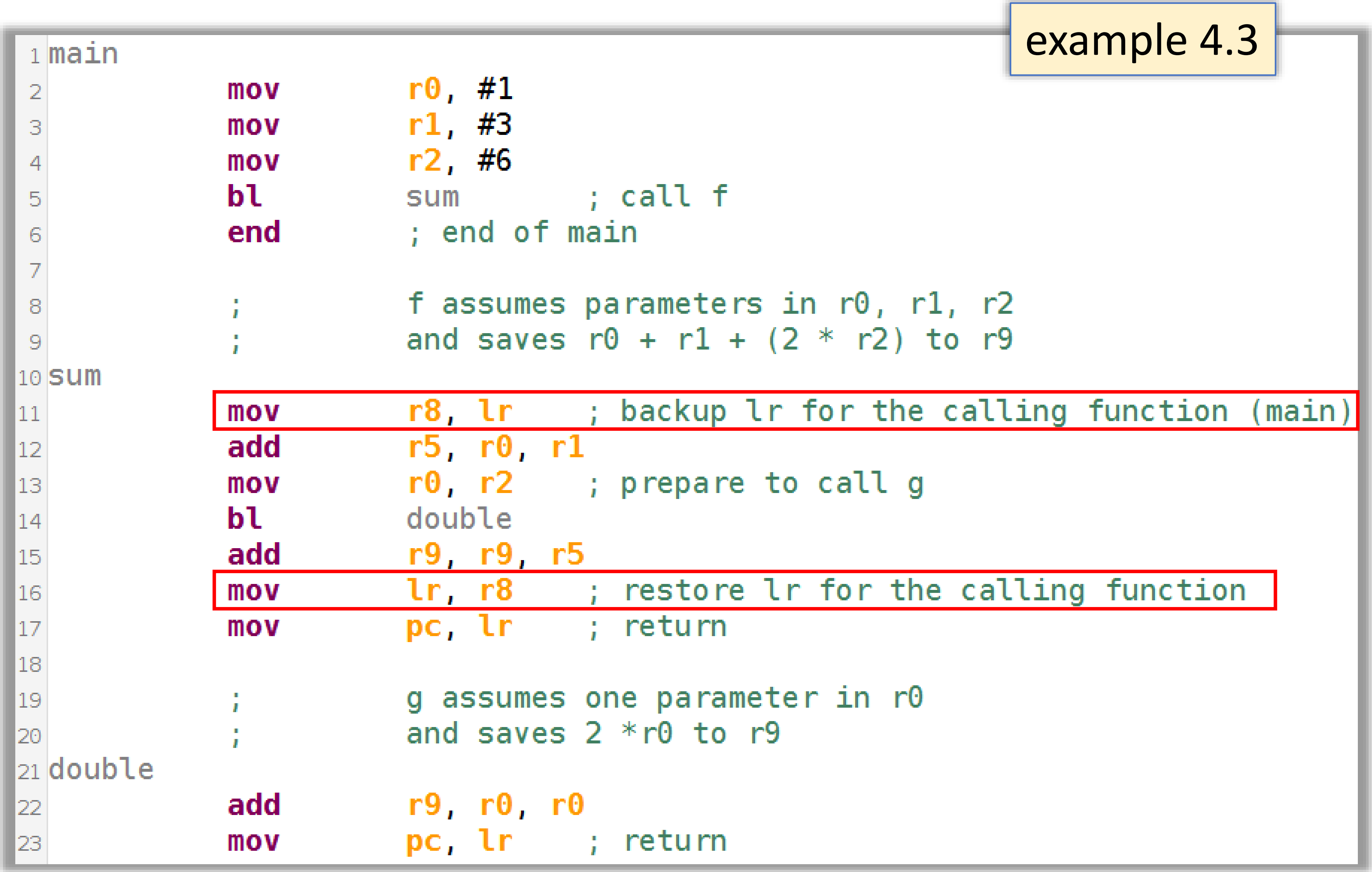
Example 4.2: Discussion

- Execution flow: main -> sum -> double -> sum -> ???
  - Run this program to find out

- The instructions `bx` and "`mov pc, lr`" do not work if a called function calls another function.
  - The register `LR` will be overwritten at the second function call.
  - The first value of `LR` will be lost.

- Workaround: backup LR to another register before calling a function and restore LR once that function returns:

example 4.3



```
main
          mov       r0, #1
          mov       r1, #3
          mov       r2, #6
          bl        sum       ; call f
          end       ; end of main

          ;         f assumes parameters in r0, r1, r2
          ;         and saves r0 + r1 + (2 * r2) to r9
sum
          mov       r8, lr    ; backup lr for the calling function (main)
          add       r5, r0, r1
          mov       r0, r2    ; prepare to call g
          bl        double
          add       r9, r9, r5
          mov       lr, r8    ; restore lr for the calling function
          mov       pc, lr    ; return

          ;         g assumes one parameter in r0
          ;         and saves 2 *r0 to r9
double
          add       r9, r0, r0
          mov       pc, lr    ; return
```

Example 4.3: Discussion

- Execution flow: main -> sum -> double -> sum -> main
  - Works properly now.

- But what if the function "double" calls another function?
  - Need another register to backup LR.

- How many registers do we have? Enough?
  - Need registers to backup LR
  - Need registers to backup parameters
  - Need registers to backup local variables of each function.
  - Registers will soon be all used.

- Need to make use of memory!
  - 16 GB vs a few 32-bit registers

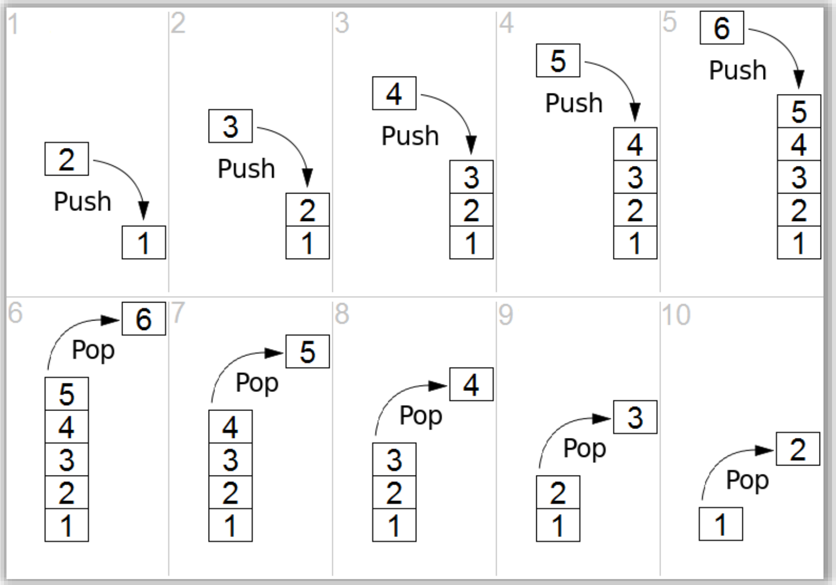
ARM Function Call

- To support multi-level function calls (also called **nested subroutines**), we need to use memory to store local variables, parameters, return addresses.
  - The memory usage grows when a function is called.
  - The memory usage shrinks when a function returns.
  - We use a data structure called **stack** to achieve this.
- Parameters are passed to the function through r0 ~ r3.
  - What if r0 ~ r3 are not enough?

- SP (r13) records the current position of the stack.
- FP (r11) is the frame pointer (not always used).
- LR (r14) records the return address.
- PC (r15) is the program counter.

The Stack

- The stack is a data structure, known as last in first out (LIFO).
- In a stack, items entered at one end and leave in the reversed order.
- Stacks in microprocessors are implemented by using a stack pointer to point to the top of the stack in memory.
  - As items are added to the stack (pushed), the stack pointer is moving up
  - and as items are removed from the stack (pulled or popped), the stack pointer is moved down.



Stack Classification

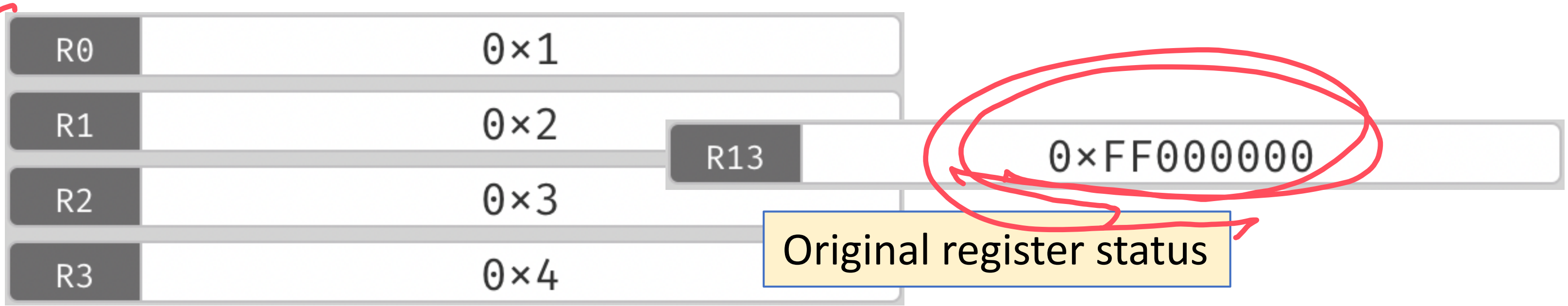
- Based on the direction of stack growth:
  - Ascending Stack - When items are pushed on to the stack, the stack pointer is increasing. That means the stack grows towards higher address.
  - Descending Stack - When items are pushed on to the stack, the stack pointer is decreasing. That means the stack is growing towards lower address.
- Based on where the stack pointer points to:
  - Empty Stack - Stack pointer points to the location in which the next/first item will be stored. e.g. A push will store the value, and increment the stack pointer for an Ascending Stack.
  - Full Stack - Stack pointer points to the location in which the last item was stored. e.g. A pop will decrement the stack pointer and pull the value for an Ascending Stack.

Stack Operation Instructions

- There are two instructions in VisUAL that support partial stack operations: STM and LDM

Syntax	Example
STM{*addr_mode*}{*cond*} *Rn*{!}, *reglist*	see example 4.4
LDM{*addr_mode*}{*cond*} *Rn*{!}, *reglist*	see example 4.4

- addr_mode can be:
  - IA: Increment address After each transfer
  - IB: Increment address Before each transfer
  - DA: Decrement address After each transfer
  - DB: Decrement address Before each transfer.
- {!}: if ! is present, the final address is written back into *Rn*.
- reglist: is a list of one or more registers to be loaded/saved



R0	0×1
R1	0×2
R2	0×3
R3	0×4

R13	0×FF000000

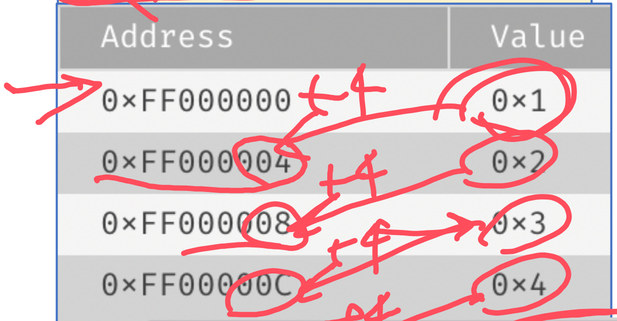
Original register status

Increment After transfer:

1. Transfer r0 to the current address, then increase SP by 4;
2. Transfer r1 to the current address, then increase SP by 4;

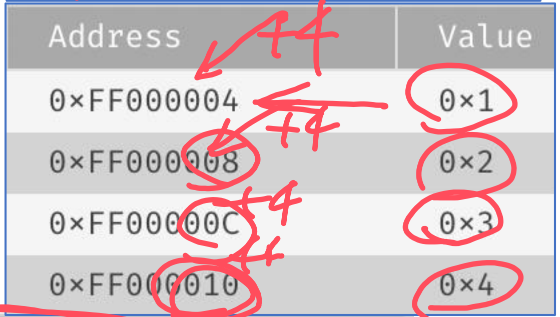
...

stmia sp!, {r0, r1, r2, r3}

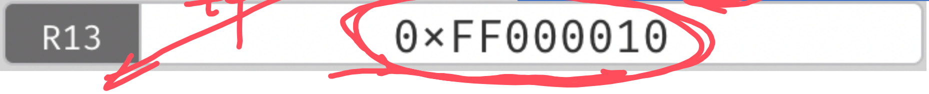


Address	Value
0×FF000000	0×1
0×FF000004	0×2
0×FF000008	0×3
0×FF00000C	0×4

stmib sp!, {r0, r1, r2, r3}



Address	Value
0×FF000004	0×1
0×FF000008	0×2
0×FF00000C	0×3
0×FF000010	0×4



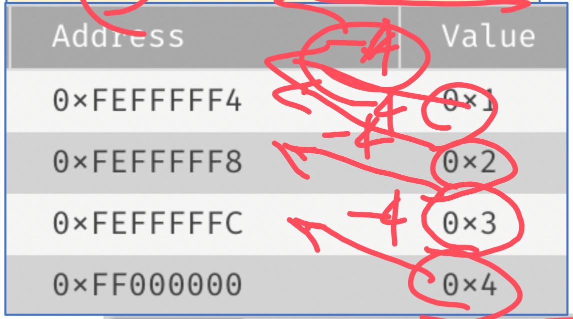
R13	0×FF000010

Decrement After transfer:

1. Transfer r3 to the current address, then decrease SP by 4;
2. Transfer r2 to the current address, then decrease SP by 4;

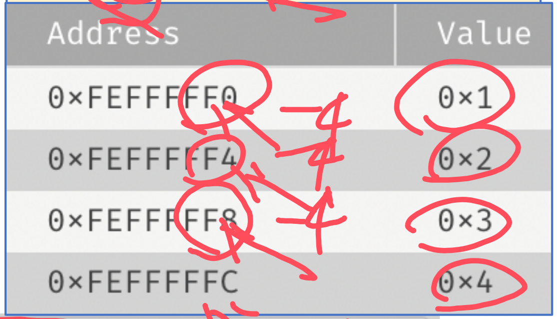
...

stmda sp!, {r0, r1, r2, r3}

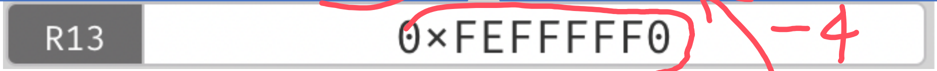


Address	Value
0×FEFFFFF4	0×1
0×FEFFFFF8	0×2
0×FEFFFFFC	0×3
0×FF000000	0×4

stmdb sp!, {r0, r1, r2, r3}



Address	Value
0×FEFFFFF0	0×1
0×FEFFFFF4	0×2
0×FEFFFFF8	0×3
0×FEFFFFFC	0×4



R13	0×FEFFFFF0

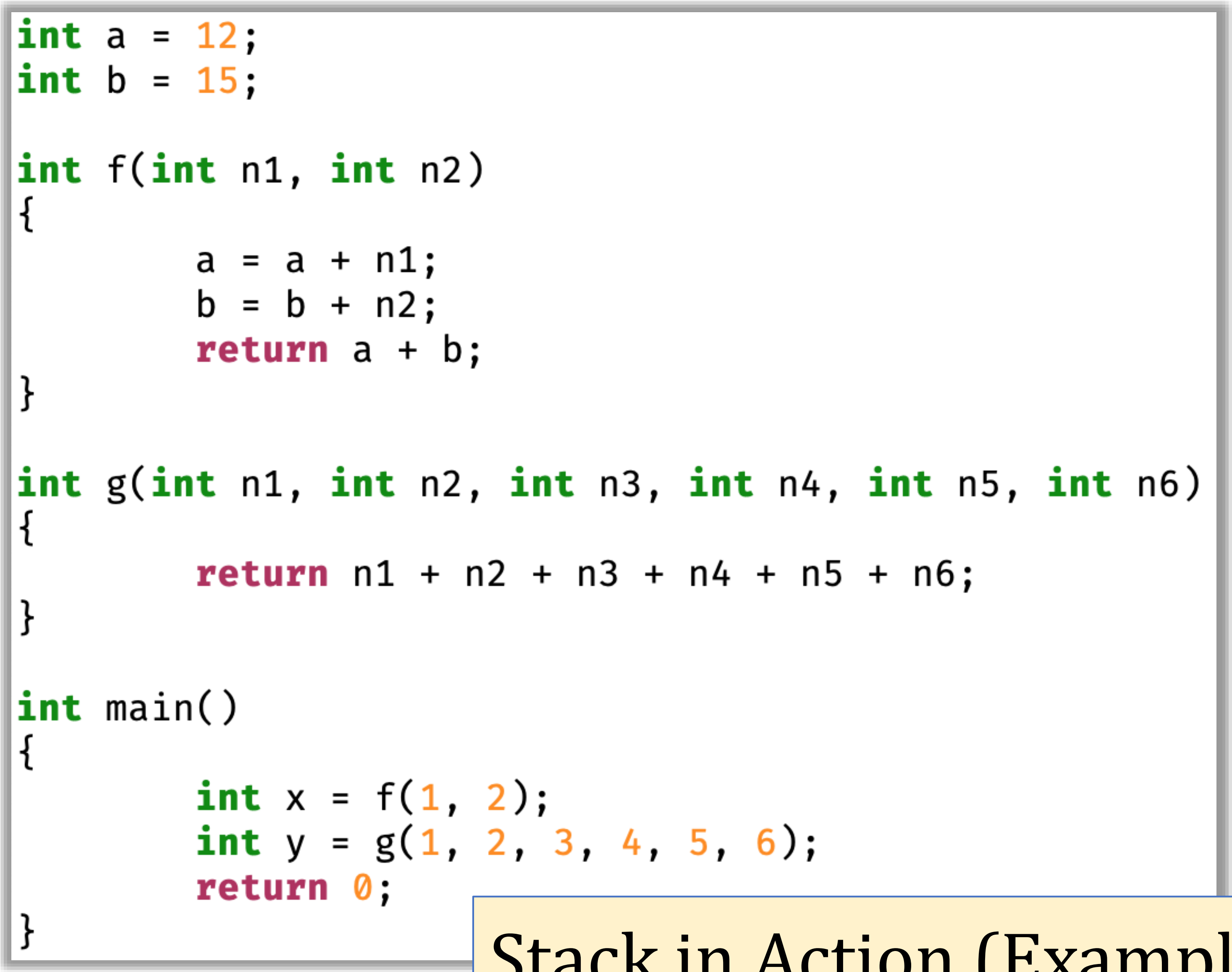
Stack Operation Instructions

- If a full descending stack is used, you can simple use the instructions: PUSH and POP (NOT supported by VisUAL)

Syntax	Example
POP{*cond*} *reglist*	pop {r7, pc}
PUSH{*cond*} *reglist*	push {r7, lr}

- `POP` has the same effect as "`LDMIA sp!, reglist`".
- `PUSH` has the same effect as "`STMDB sp!, reglist`".

Stack in Action (Example 4.5)



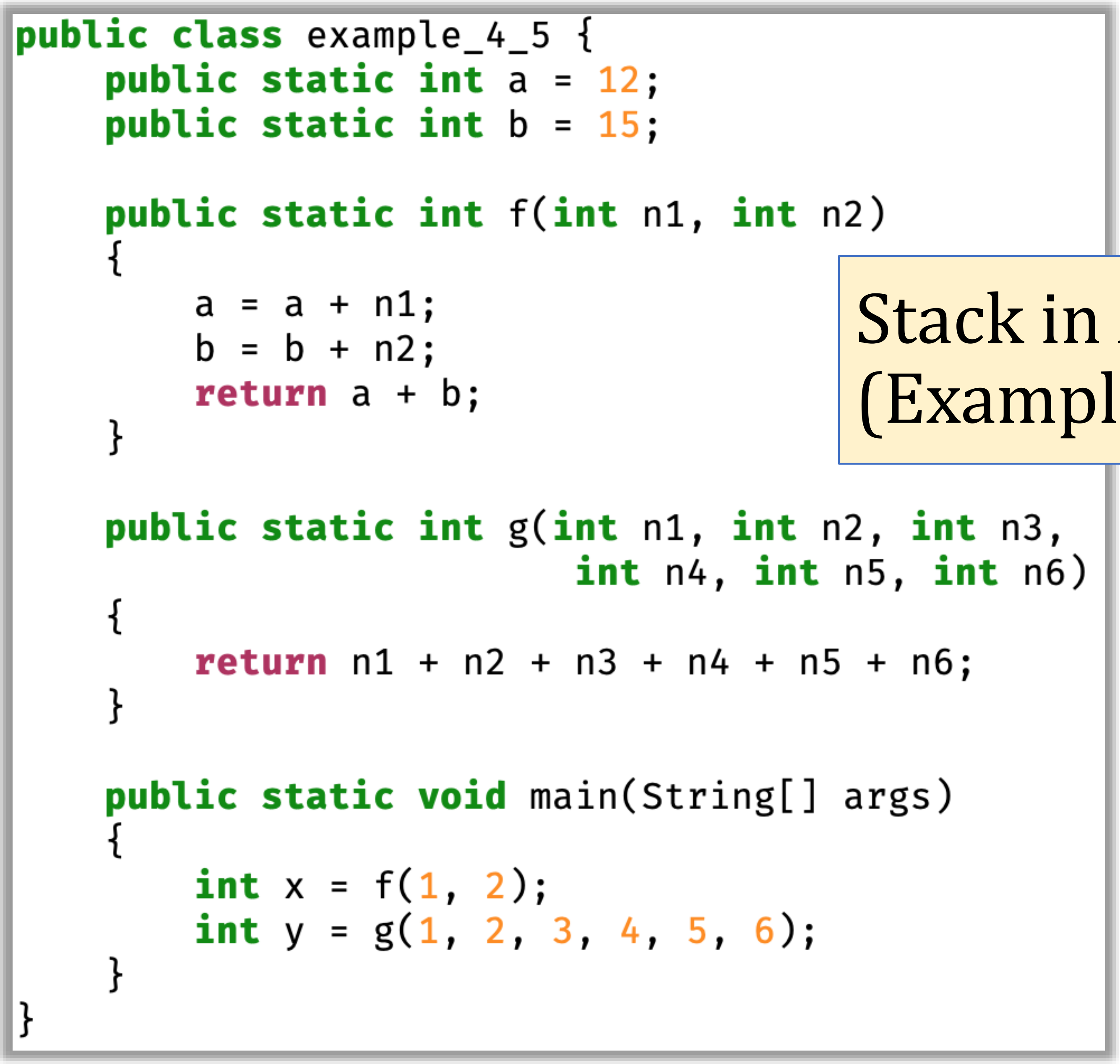
```c
int a = 12;
int b = 15;

int f(int n1, int n2)
{
    a = a + n1;
    b = b + n2;
    return a + b;
}

int g(int n1, int n2, int n3, int n4, int n5, int n6)
{
    return n1 + n2 + n3 + n4 + n5 + n6;
}

int main()
{
    int x = f(1, 2);
    int y = g(1, 2, 3, 4, 5, 6);
    return 0;
}
```

Stack in Action (Example 4.5 Java)



```java
public class example_4_5 {
    public static int a = 12;
    public static int b = 15;

    public static int f(int n1, int n2)
    {
        a = a + n1;
        b = b + n2;
        return a + b;
    }

    public static int g(int n1, int n2, int n3,
                        int n4, int n5, int n6)
    {
        return n1 + n2 + n3 + n4 + n5 + n6;
    }

    public static void main(String[] args)
    {
        int x = f(1, 2);
        int y = g(1, 2, 3, 4, 5, 6);
    }
}
```

Example 4.5: Discussion

Look at the C or Java code and compare them against their corresponding assembly code. Answer questions below:

- Question 1: How are the parameters of function g passed from main?
- Question 2: How are the return values of f and g passed to main?
- Question 3: Can you map every line of the function f to its assembly code?
- Question 4: Where are the local variables of f and g stored?
- Question 5: Where are variables a and b stored?

Important Things about Functions

- Each function has a stack frame, which holds all local variables (including parameters) of that function
  - What is the stack frame of the function f?
- Global variables are not stored inside stack.
  - The “.word” you see in the example is similar to “DCD” we used before. They are followed by the values of these global variables.
- Some more information: https://www.techopedia.com/definition/22304/stack-frame#:~:text=A%20stack%20frame%20is%20comprised%20of%3A%201%20Local,need%20restoration%203%20Argument%20parameters%204%20Return%20address

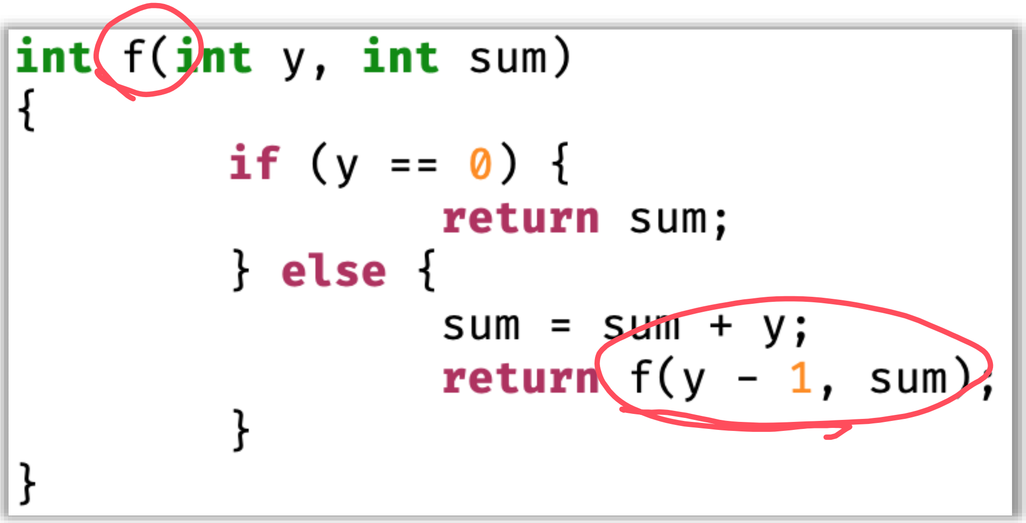
Case Study: Recursive Function Calls in Assembly

And tail call elimination in action

Concept 1: Recursive Function Calls

- Recursive function: A function “that calls itself one or more times until a specified condition is met at which time the rest of each repetition is processed from the last one called to the first”.

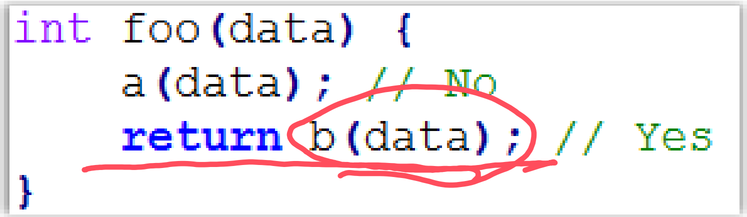
- Below is an example of recursive function (example 4.6):



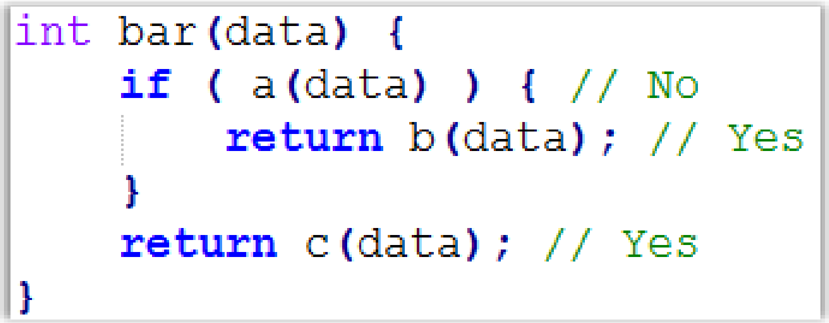
```
int f(int y, int sum)
{
        if (y == 0) {
                return sum;
        } else {
                sum = sum + y;
                return f(y - 1, sum);
        }
}
```

Concept 2: Tail Call and Tail-Recursive

- A **tail call** is a function call performed as the final action of a procedure.



```
int foo(data) {
    a(data); // No
    return b(data); // Yes
}
```

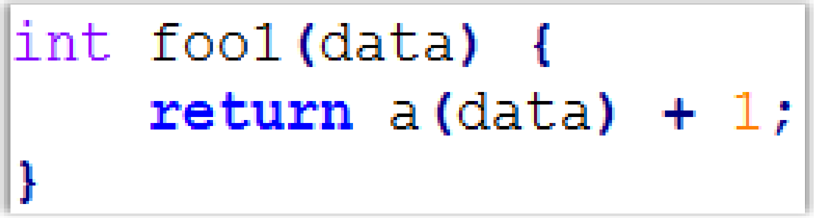


```
int bar(data) {
    if ( a(data) ) { // No
        return b(data); // Yes
    }
    return c(data); // Yes
}
```

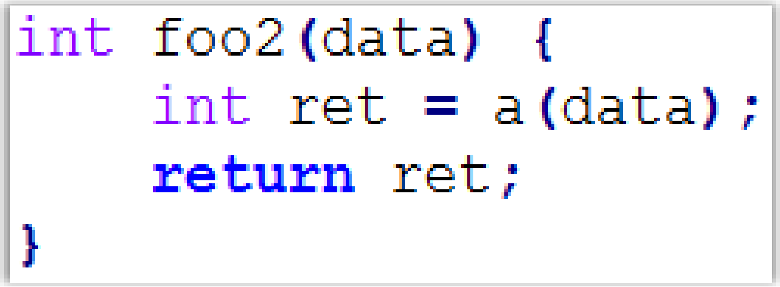
- If a tail call might lead to the same function being called again later in the call chain, the function is said to be **tail-recursive**.
    - Example 4.6 is tail-recursive

Tail Call: More Examples

- Do the following examples involve tail calls?



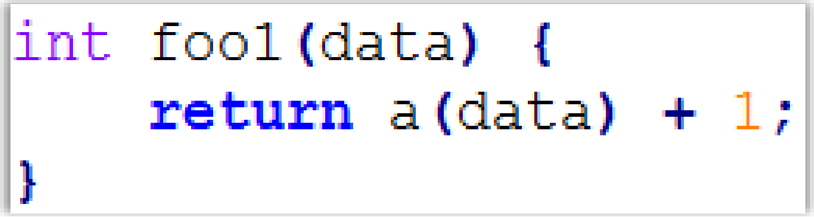
```
int foo1(data) {
    return a(data) + 1;
}
```



```
int foo2(data) {
    int ret = a(data);
    return ret;
}
```

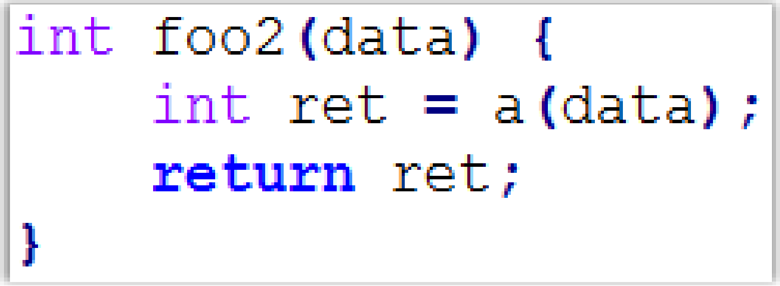
Tail Call: More Examples

- Do the following examples involve tail calls?



```
int foo1(data) {
    return a(data) + 1;
}
```

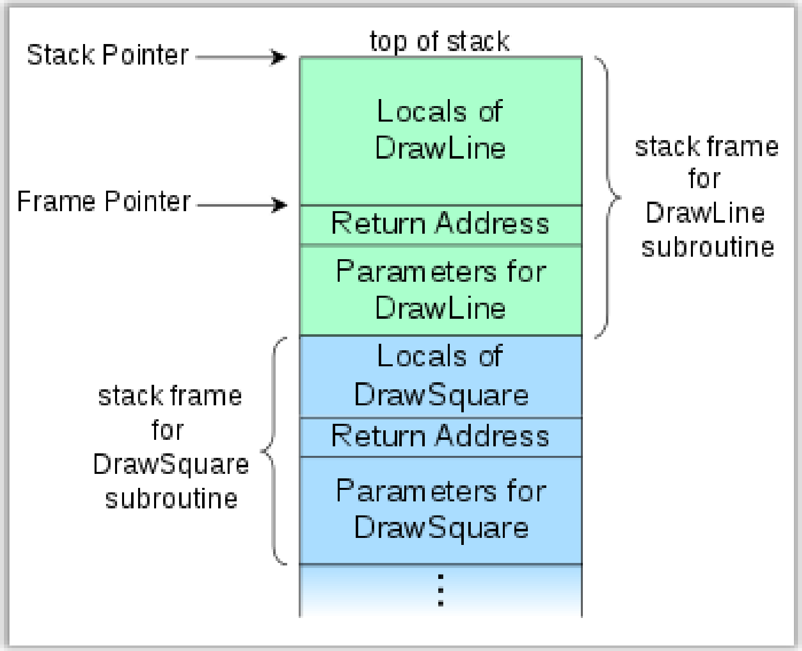
No!



```
int foo2(data) {
    int ret = a(data);
    return ret;
}
```

Tail Call Optimisation

- If you let a function call itself for many times, it will eventually consume too much memory that the operating system will kill the program.
- For tail calls, they can be implemented without adding a new stack frame to the **call stack**.
- This optimisation technique is called tail call elimination.
  - Tail recursions can be as efficient as `for` loops



Example 4.6

- The example 4.6 has two pieces of associated assembly code.
  - One is compiled normally.
  - Another has tail call optimisation on.

- Question 1: Can you map every line of the function to its assembly (both versions)?

- Question 2: This is a design problem, why it is safe to not “adding a new stack frame to the call stack”?