Computer Systems
Lecture 13

Overview

- Subroutines
- Return addresses
- Subroutines in assembly language
- Return from subroutine call
- How does CALL work
- Nested calls
- CALL and RET working together
- Using the stack

Subroutines

- We have seen many higher level programming language constructs can be implemented at a lower level.
- Time to consider the implementation details of another important feature of HLLs, that is, the **subroutine mechanism**.

Subroutines (cont.)

- A **subroutine** is a general term. In different programming languages it may be called differently:
  - **Procedure** in Pascal.
  - **Function** in C.
- Yet the idea is the same: a subroutine is a part of the code, which can be used repeatedly within the program that is being executed.

Subroutines: an example

```
…
lea eax, input
push eax
lea eax, format     // address of the format string is saved in eax
push eax            // push the address of the string to the stack
call scanf          // scanf("%d",&input);

add esp,8           // clean top two positions in the stack
push input          // push the value of the input onto the stack
lea eax,message     // address of the message string is saved in eax
push eax            // the value of eax is pushed onto the stack
call printf         // prinf("%d", input);

add esp, 8          // clean top two positions in the stack
…
```

- In the above example two subroutines were used: **`scanf`** and **`printf`**.
- **`Call`** is the instruction to call a subroutine.

Subroutines: what are they good for?

- Save the effort in programming. (Why?)
- Reduce the size of programs. (Why?)
- Share the code. (How?)
- Encapsulate, or package, a particular activity.
  - Hide complications from the user.
- Provide easy access to tried and tested code.
- Facilitate code reuse.

Fundamental issue: where did I come from?

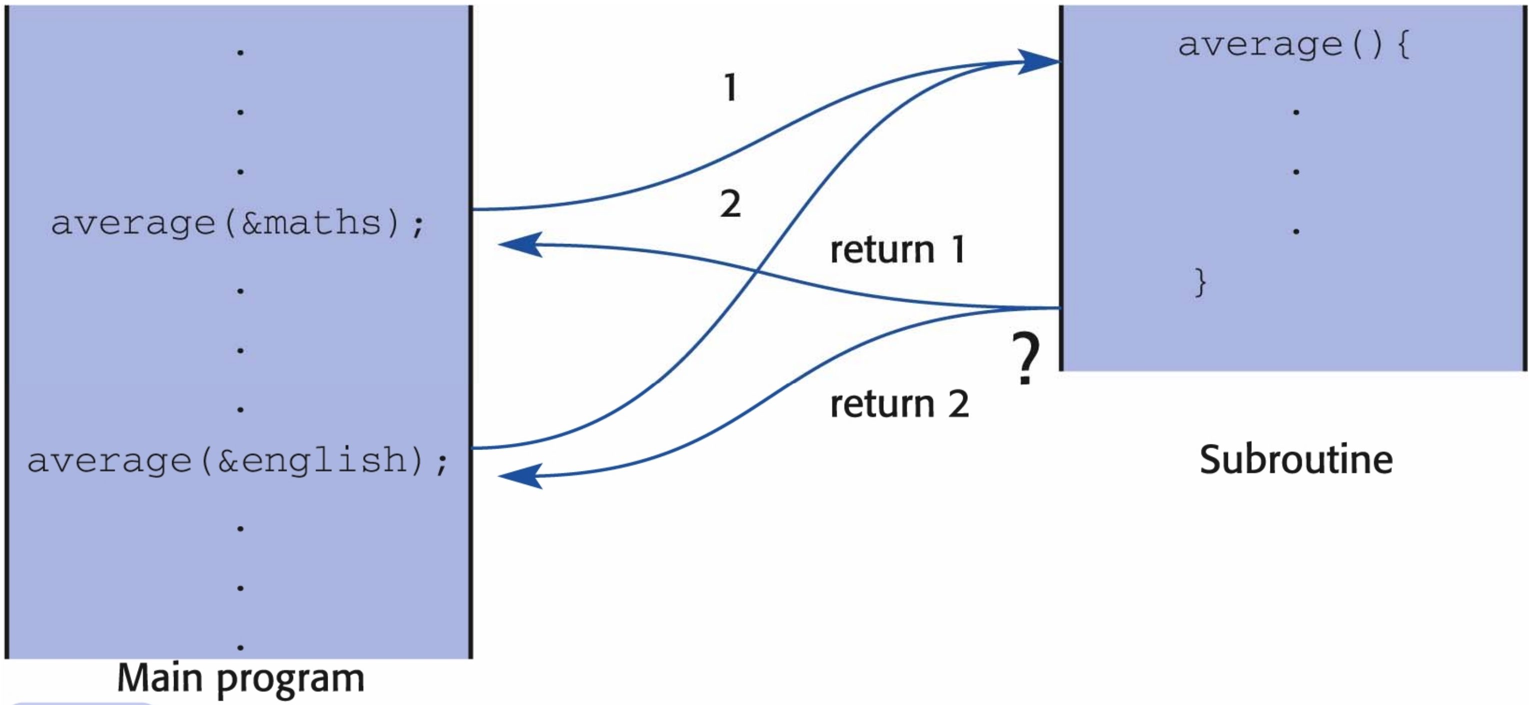
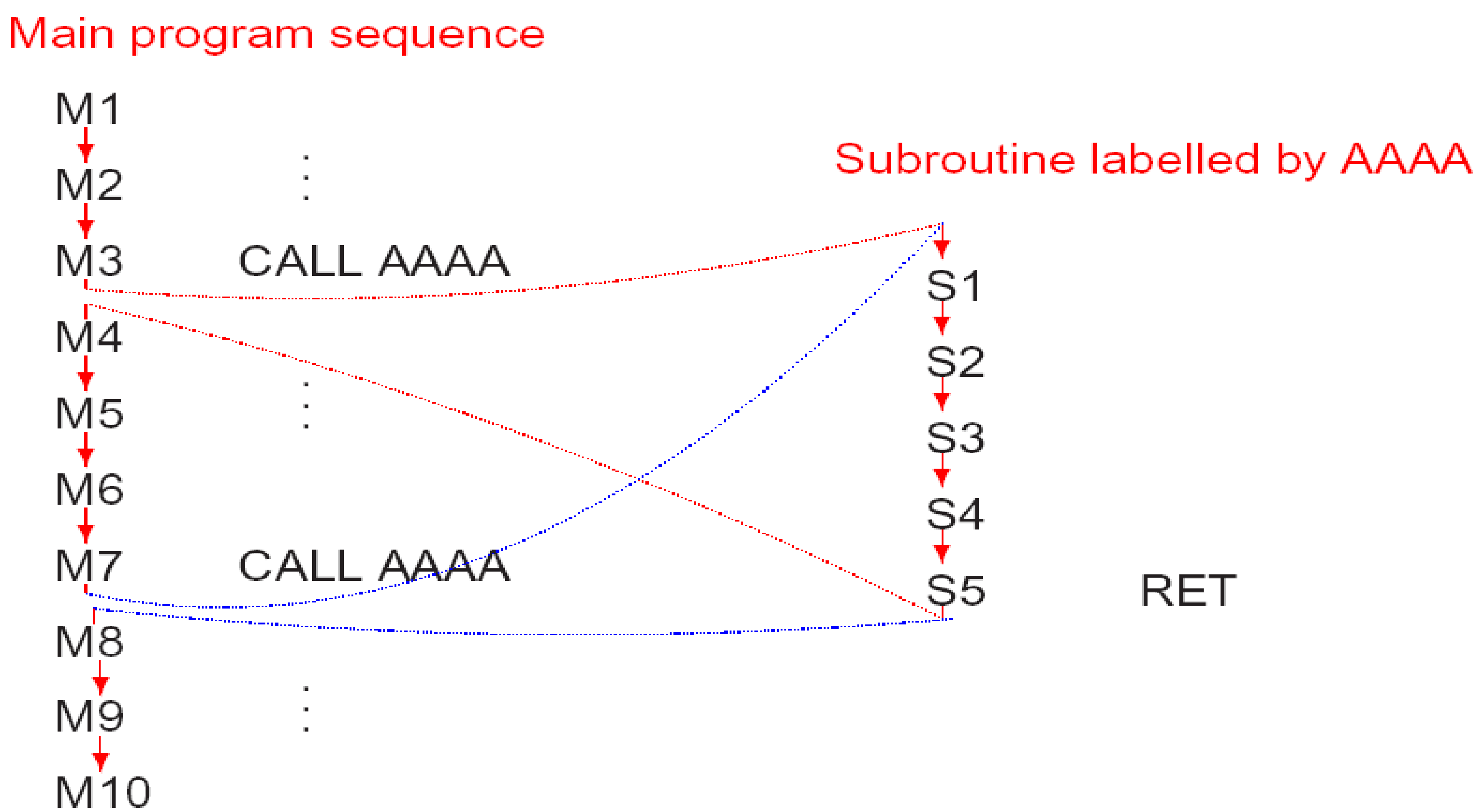


Fig. 8.1 Where did I come from?

- The subroutine is invoked at different moments from two different places in the main program.

Subroutines in assembly language



Return addresses

- How does the computer know where to return from a subroutine?
- Different calls of the same subroutine return to different places.

- One needs to store a **return address** for every call of the subroutine.

- Where is the return address stored?
- How much room do we need?

Subroutines in assembly language

- Before we answer the above questions, let us look at how a subroutine can be declared and used in the (MASM) assembly language.

- Declaration of a simple procedure looks as follows:

```
label PROC
        …
        …
        …
        RET             ; return
label ENDP
```

- The procedure can be called by the instruction:

```
call label
```

Return from the subroutine

- The instruction RET changes the control, causing execution to continue from the point following the CALL instruction.

How does CALL work

- CALL does the following:
  - Records the current value of EIP (Instruction Pointer) as the **return address**.
  - Places the required subroutine address into EIP (instruction Pointer) , so the next instruction to execute is the first instruction of the subroutine.

Nested calls

```
      . . .
        :
      CALL SUB1       * call first subroutine
        :
SUB1: . . .
        :
      CALL SUB2       *call second subroutine
        :
      RET
SUB2: . . .
        :
      RET
```

- If the return address was stored in a dedicated memory location, it would work for the first call.
- But when it comes to the second call, we would have no place to store the return address.

Nested calls and return addresses

- In the last example we need **both return addresses** stored **at the same time**. (Why?)
- The more deep nesting of subroutine calls, the more space do we need to store all return addresses.

- An elegant solution is to use a **stack** to store all return addresses.

CALL and RET working together

- CALL does the following:
  - Records the current value of EIP (Instruction Pointer) as the **return address**: copy the address from EIP to the stack (PUSH).
  - Puts the required subroutine address into EIP (Instruction Pointer) , so the next instruction to execute is the first instruction of the subroutine.
- RET does the following:
  - Pop the last address stored in the stack and put it into EIP.

Using the stack

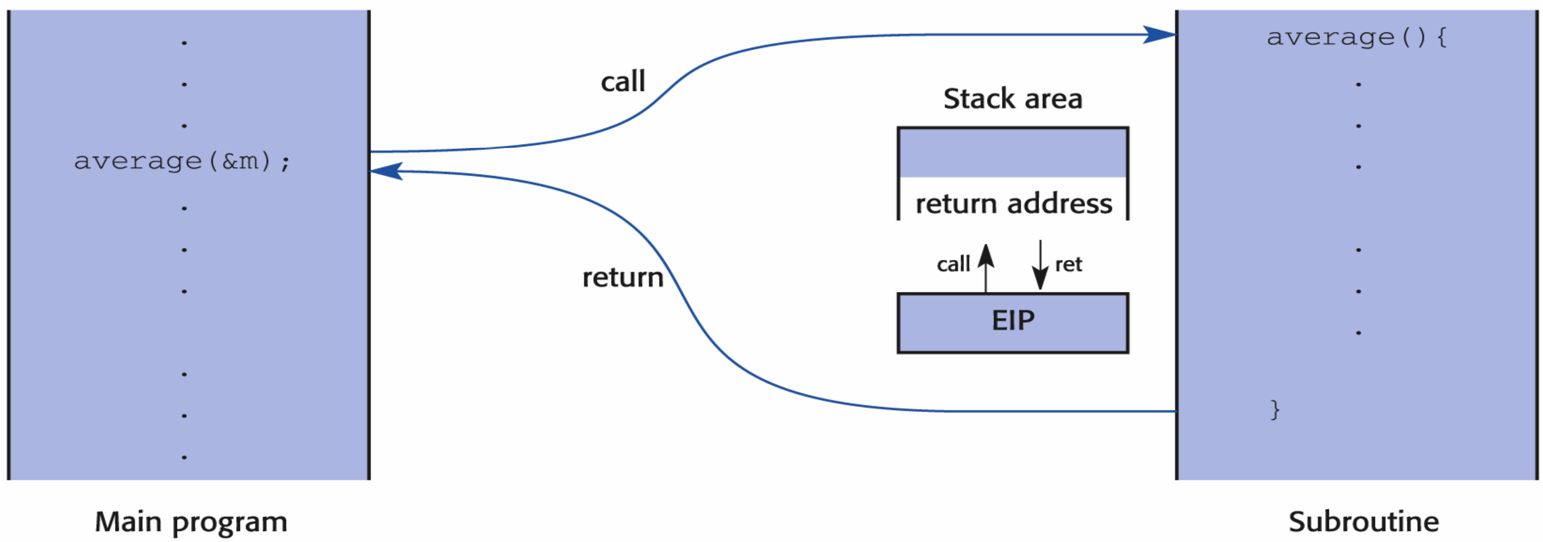


Fig. 8.3 Using the stack to hold the subroutine return address.

- Q. A subroutine can be called at various moments from different places in the main program. (T or F)

- Q. A subroutine can be called by itself. (T or F)

- Q. How many return addresses can you store using a stack?

- Q. How many nested calls can you make within a program?

- Q. How does the computer know when to return from a subroutine?

- Q. What happens when a ‘CALL …’ instruction is executed?

- Q. What happens when a ‘RET’ instruction is executed?

Readings

- [Wil06] Sections 8.1, 8.2, 8.3.