Computer Systems
Lecture 14

Overview

- Value parameters
- Reference parameters
- Passing parameters via registers
- Local variables
- Passing parameters via stack
- Nested subroutine calls

Parameters

- The simplest kind of subroutine performs an identical function each time it is called. Such a subroutine requires no further information to run.
- But most subroutines require additional information, given in **parameters**.
  - For example, the **`printf`** subroutine requires information re what to print and how to print.
- Stack can be used to pass parameters to subroutines (lecture 11), and to store return addresses of subroutine calls (lecture 13). How does all this work together?

Parameters (cont.)

- In general, parameters can take a number of forms, depending on the nature of the information required by the subroutine.
- Two main forms of parameters:
  - Value parameters.
  - Reference parameters.

Value parameters

- In many cases, the additional information required by a subroutine will be a simple **value** (or values) of some type(s):
  - For example, a numeric value, or ASCII code value, etc.
- Imagine a subroutine, when given two numbers, has to decide and return which number is bigger. All this subroutine needs is the values of two variables.
- Such parameters are called **value parameters**.

An example: value parameters

```
        …
        mov eax, first          ;
        mov ebx, second         ;
        call bigger             ; calling
        mov max, eax            ;
        …
bigger  proc                    ; procedure which uses value parameters
        mov save1, eax          ; passed through registers EAX and EBX
        mov save2, ebx          ;
        sub eax, ebx            ; the result of the procedure is returned
                                ; again via register EAX
        jg first_big            ;
        mov eax, save2          ;
        ret                     ;
first_big: mov eax,save1        ;
        ret                     ;
bigger  endp                    ;
```

Exercise

- Write a subroutine: when given two numbers in EAX and EBX, it returns their difference via EAX.

Exercise

- Write a subroutine: when given two variables, it swaps their values.

Reference parameters

- Consider the following subroutine: “Given two variables, swap their values”.
- Having only the values of variables is not enough. We need another type of parameters here.
- A variable is a location in the memory, the name of the variable determines the address of this location.
- Thus, it will be enough for the above procedure to have the addresses of the variables as an additional information.
- Such kind of parameters are called **reference parameters**.

An example: reference parameters

```
      …
      lea eax, first          ;
      lea ebx, second         ;
      call swap               ; calling
finish: …
  ;
swap proc                     ; subroutine which uses
      mov temp, [eax]         ; reference parameters
      mov [eax],[ebx]         ; passed via registers
      mov [ebx],temp          ;
      ret                     ;
swap endp
```

Passing parameters via registers

- The simplest method of passing parameters into a subroutine is to use a register:
  - Copy a value into an available CPU register and then jump to the subroutine.
- Both value and reference parameters can be passed using registers.
- But, this method would be too constraining!
- Why?

Stack instead of registers

- For subroutines with a **few** parameters one may pass these parameters (in address, or value form) using general-purpose registers.
- However, in general, the number of registers available is very limited, and is not enough to implement subroutines with an arbitrarily large number of parameters.
- Thus, we need to use the memory in order to implement parameter handling for subroutines.
- The solution in most computers is to use the **stack** for passing parameters and keeping the local variables.

Local variables

- In the example with value parameters the instructions make use of two variables, **`save1`** and **`save2`** as a temporary storage for parameters.
- They may be thought of as local variables of the subroutine, in contrast to the global variables used throughout the program.
- Where are they stored during the execution of the subroutine?

Stack frame

- The area of the stack which holds all data related to one subroutine call is called a **stack frame**.
- This data includes:
  - Parameters of the subroutine.
  - Return addresses.
  - Local variables.

EBP and ESP for stack frames

- Because of the nested calls several stack frames may be present at the same time.
- Two registers are used to work with stack frames:
  - **EBP**: The stack frame base pointer.
    - To indicate the base of the current stack frame.
  - **ESP**: The stack pointer.
    - To hold the address of the top of the stack.

How does all this work?

- ESP is always pointing to the top of the stack.
- EBP initially contains an address of the base of the stack.

- Just before and during the call of a subroutine the following happens:
  - The parameters are pushed on the stack.
  - The return address is pushed on the stack.
  - The address stored in EBP is pushed on the stack.
  - A new stack frame is created.
  - The current address of the top of the new stack frame is saved in EBP.
  - The local variables are installed on the new stack.

How does all this work? (Cont.)

- Once the subroutine done its job:
  - Pop all local variables out of the stack.
  - Pop the previous EBP address from the top of the stack and restore it in EBP (stack frame base pointer).
  - Clean up parameters in the stack.
  - Pop the return address and save it in EIP.
  - Note: popping order is crucial.

Summary: a scenario of nested subroutine calls

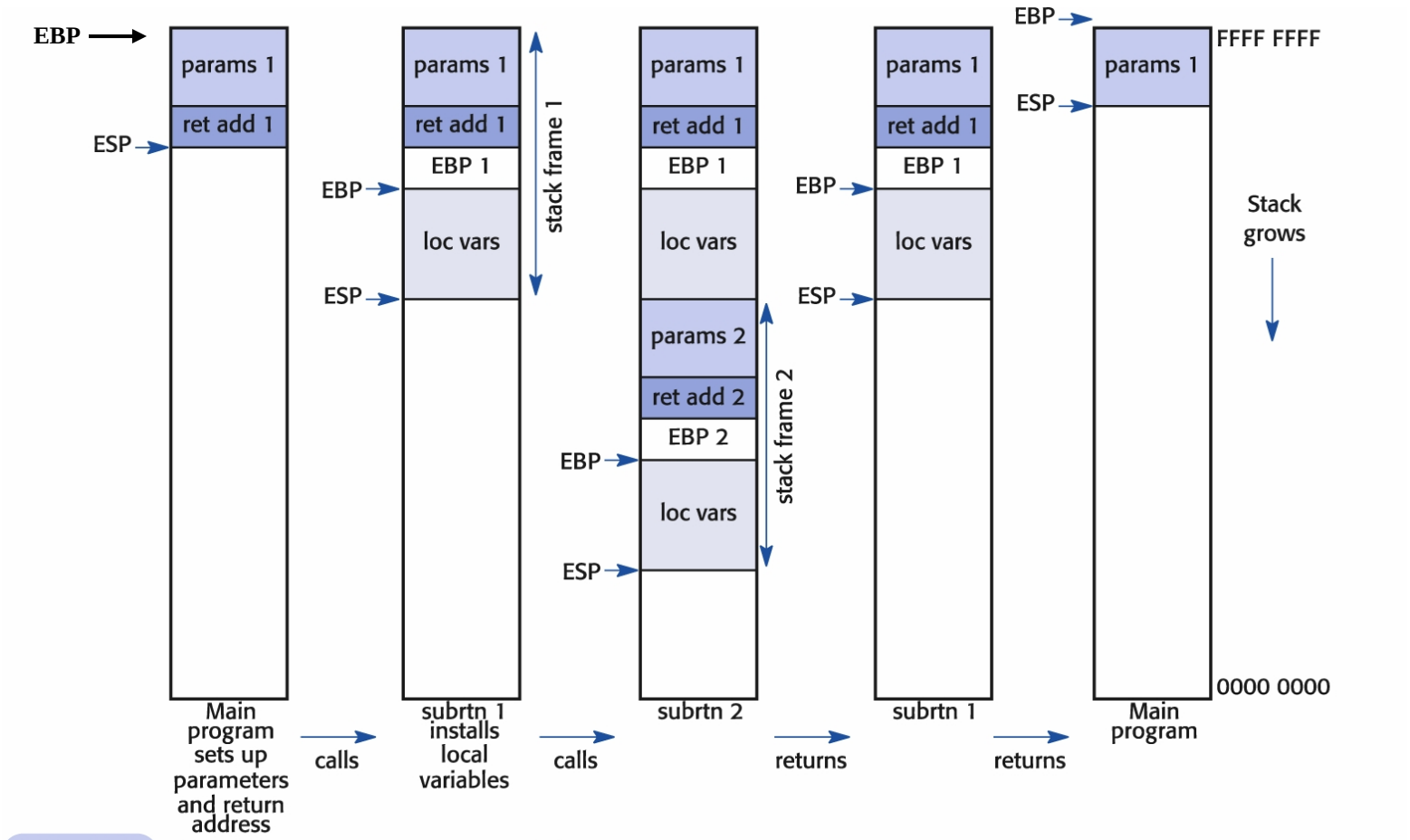

Fig. 8.9 Using thestack for local variables.

- Q. When is value parameter good for?

- Q. When is reference parameter good for?

- Q. Why multiple stack frames can coexist at the same time?

- Q. What type of data are stored under a stack frame?

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

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

Readings

- [Wil06] Chapter 8, sections 8.5, 8.6.