SISTEMI EMBEDDED

Stack, Subroutine, Parameter Passing
C Storage Classes and Scope

Federico Baronti

Last version: 20160314

Stack

- A stack is an abstract data structure managed according to a last-in-first-out (LIFO) policy
- Consists of a sequential collection (list) of data elements, where elements are added/removed at top end only
- We push a new element on the stack top or pop the top element from the stack
- Programmer can create a stack in the memory
- There is often a special processor stack as well

Processor Stack

- Processor has the stack pointer (SP) register that points to top of the processor stack
- Push operation involves two instructions:

```
Subtract    SP, SP, #4
Store       Rj, (SP)
```

- Pop operation also involves two instructions:

```
Load    Rj, (SP)
Add     SP, SP, #4
```

Subroutine (1)

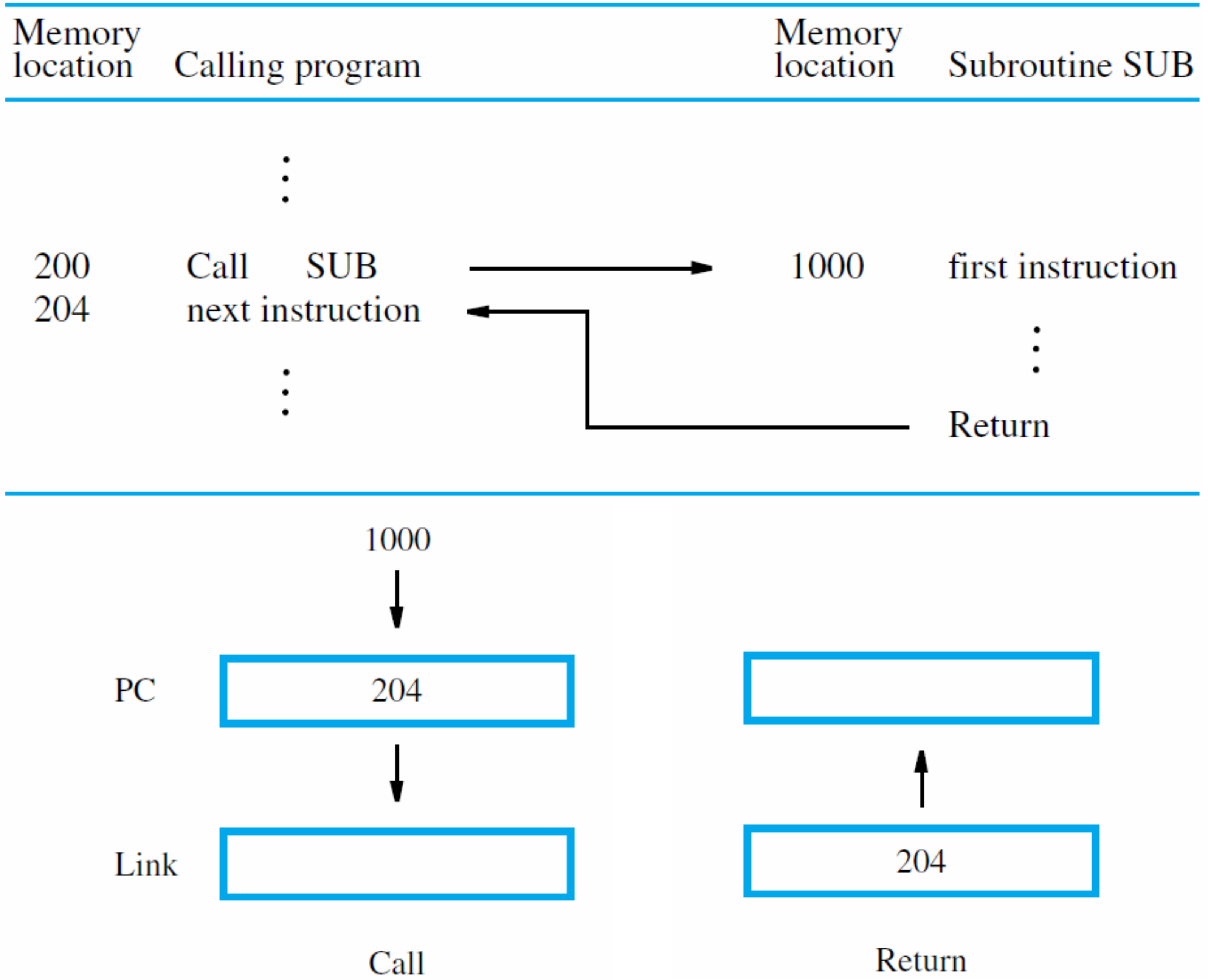
- In a given program, a particular task may be executed many times using different data
  - Support for structured and modular coding style
- A subroutine implement task in one block of instructions
- Rather than reproducing the entire subroutine block in each part of program that uses it, exploits a subroutine Call
  - However, in C language, we may force the compiler to copy the subroutine code in each point is called by declaring the subroutine with the **inline** keyword
- A Call instruction is a special type of branch

Subroutine (2)

- Branching to same block of instructions saves space in memory (single copy of the subroutine block), but we must implement a mechanism to branch back
  - The subroutine must return to the calling program after executing last instruction in subroutine
- This branch is done with a **Return** instruction
- Subroutine can be called from different places
- How can return be done to the correct place?
  - This is the issue of subroutine linkage

Subroutine Linkage

- During execution of Call instruction, PC is firstly updated to point to the instruction after Call
- Save this address to be used by the Return instruction in the called subroutine
- Simplest method: place address in a link register
- Call instruction thus performs two operations:
  - store updated PC contents in the link register,
  - then branch to target (subroutine) address
- Return just branches to address in link register


Memory location
Calling program
Memory location
Subroutine SUB
200
Call SUB
204
next instruction
1000
first instruction
Return
1000
PC
204
Link
204
Call
Return

Subroutine Nesting and the Stack

- We can permit **only** one subroutine to call another using just the link register.
  - Link register contents after first subroutine call are overwritten after second subroutine call
- To allow **subroutine nesting**, first subroutine should save link register on the processor stack before second call
- After return from second subroutine, first subroutine restores the link register from the processor stack

Parameter Passing

- A program may call a subroutine many times with different data to obtain different results
- Information exchange to/from a subroutine is called parameter passing
- Parameters may be passed in registers
- Simple, but limited to available registers
- Alternative: use stack for parameter passing, and also for local variables & saving registers

Calling program

```
                Load R2, N                    Number of element to be added
                Move R4, #NUM1                Address of the first element
                Call LISTADD
                Store R3, SUM                 Result of the LISTADD subroutine
                ...
```

Subroutine

```
LISTADD:        Subtract SP, SP, #4
                Store R5,   (SP)              Save R5 in the stack
                Subtract SP, SP, #4
                Store R2,   (SP)              Save R2 in the stack
                Subtract SP, SP, #4
                Store R4,   (SP)              Save R4 in the stack
                Clear R3
LOOP:           Load R5, (R4)
                Add R3, R3, R5
                Add R4, R4, #4
                Subtract R2, R2, #1
                Branch_if_[R2]>0 LOOP
                Load R4, (SP)                 Restore R4 from the stack
                Add SP, SP, #4
                Load R2, (SP)                 Restore R2 from the stack
                Add SP, SP, #4
                Load R5, (SP)                 Restore R5 from the stack
                Add SP, SP, #4
                Return
```

Example using registers

Calling program

```
          Load R2, N              Number of element to be added
          Subtract SP, SP, #4
          Store R2, (SP)
          Move R2, #NUM1          Address of the first element
          Subtract SP, SP, #4
          Store R2, (SP)
          Call LISTADD
          Store R3, SUM           Result of the LISTADD subroutine
          Add SP, SP, #8
          ...
```

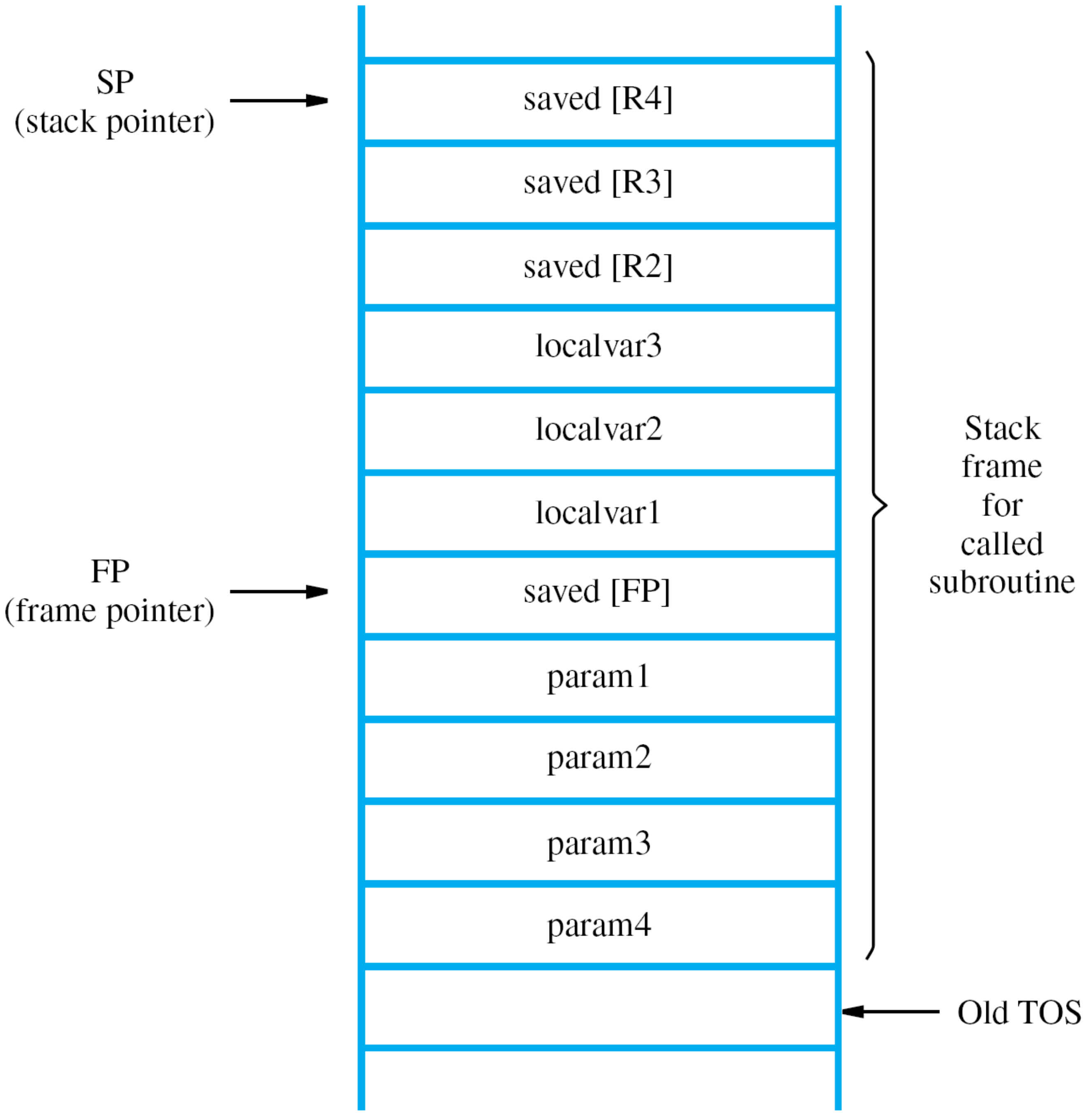
Subroutine

```
LISTADD:  Subtract SP, SP, #12
          Store R2,   8(SP)
          Store R4,   4(SP)
          Store R5,   0(SP)
          Load R4,   12(SP)
          Load R2,   16(SP)
          Clear R3
LOOP:     Load R5, (R4)
          Add R3, R3, R5
          Add R4, R4, #4
          Subtract R2, R2, #1
          Branch_if_[R2]>0 LOOP
          Load R2, 8(SP)
          Load R4,  4(SP)
          Load R5,  0(SP)
          Add SP, SP, #12
          Return
```

Example using stack

Stack Frame

- Locations at the top of the processor stack are used as a private work space by subroutines
- A stack frame is allocated on subroutine entry and deallocated on subroutine exit
- A frame pointer (FP) register enables access to private work space for current subroutine instance
  - FP facilitates access to parameters (with positive indexes) and local variable (with negative indexes)
  - FP of the caller must be saved in the stack frame
- With subroutine nesting, the stack frame also saves return address of the caller


SP
(stack pointer)
saved [R4]
saved [R3]
saved [R2]
localvar3
localvar2
localvar1
FP
(frame pointer)
saved [FP]
param1
param2
param3
param4
Stack frame for called subroutine
Old TOS

C Storage Classes and Scope (1)

- The storage class determines how long an object is kept in memory during the program execution
- A scope specifies the part of the program in which a variable name is visible, that is, the accessibility of the variable by its name
- In C program, there are four storage classes: automatic, register, external, and static.

C Storage Classes and Scope (2)

- Variables defined at the beginning of a block ({…}, e.g., at the beginning of a function) belong by default to the automatic class
  - Memory is allocated automatically upon entry to a block and freed automatically upon exit from the block
  - The scope of automatic variables is local to the block in which they are declared, including any blocks nested within that block. For these reasons, they are also called **local variables**
  - The programmer can suggest to the compiler that particular automatic variables should be allocated to CPU registers, if possible, using the register keyword in the variable definition

C Storage Classes and Scope (3)

- Variables defined outside any block are external and static. They are accessible from within any block, are created at the start of the program and live up to the program end. Such variables are called **global variables**
  - If not explicitly specified, global variables are initialized to 0.
  - The scope of external variables is global. All functions following the declaration may access the external variable by using its name. However, if a local variable having the same name is declared within a function, references to this name will access the local variable

C Storage Classes and Scope (4)

- Variables inside a block are automatic by default, but can be made static using the static keyword in their definition
  - Static variables may be initialized in their definitions
  - However, the initializers must be constant expressions, and initialization **is done only once** at compile time when memory is allocated for the static variable

C Storage Classes and Scope (5)

- To use a **global variable** in a function defined in a different file, the latter must contain the declaration only of this variable
  - This is done using the extern keyword

```
//file1.c
#include ”file2.h”

int main() {
  func2(10);
  fprint(“gvar1 = %d”, gvar1);
  //…
}
```

```
//file2.c
#include ”file2.h”

int gvar1 = 0;

void func2(int arg1) {
  gvar1 += arg1;
}
```

```
//file2.h
#ifndef FILE2_H_
#define FILE2_H_

extern int gvar1;

void func2(int arg);

#endif
```

References

- C. Hamacher, Z. Vranesic, S. Zaky, N. Manjikian "Computer Organization and Embedded Systems," *McGraw-Hill International Edition*
  – Cap. II 2.5, 2.6 and 2.7