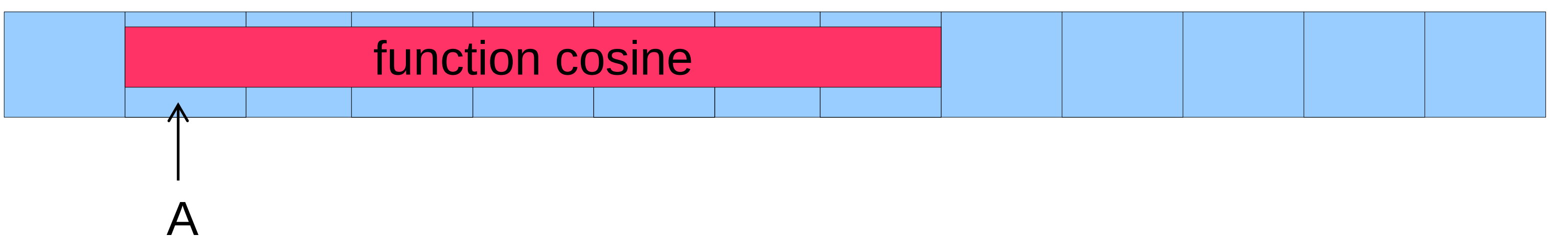
Functions and Stacks

Slide courtesy: Smruti Ranjan Sarangi

Slides adapted by: Dr Sparsh Mittal

Implementing Functions

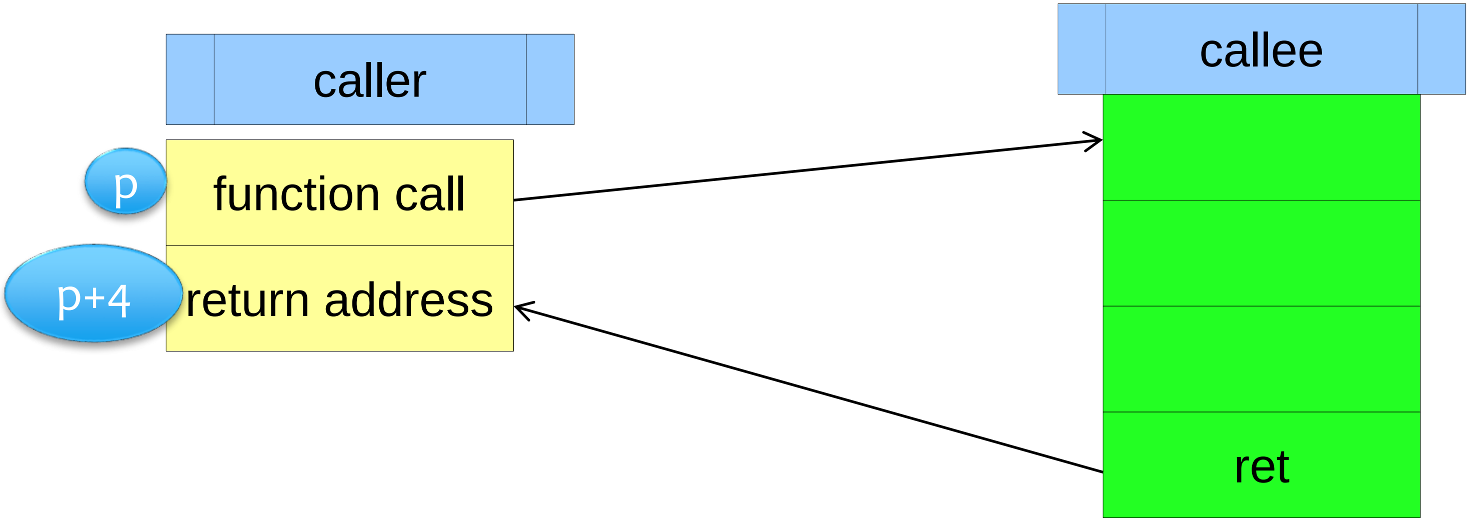
- Functions are blocks of assembly instructions that can be repeatedly invoked to perform a certain action
- Every function has a starting address in memory (e.g. cosine has a starting address A



Implementing Functions - II

- To call a function, we need to set :
  - pc $\leftarrow$ A
- We also need to store the location of the pc that we need to come to after the function returns
- This is known as the return address
- We can thus call any function, execute its instructions, and then return to the saved return address

Notion of the Return Address



- PC of the call instruction → p
- PC of the return address → p + 4

because, every instruction takes 4 bytes

How to pass arguments/ return values

* Solution : use registers

SimpleRisc

```
.func:
    add r2, r0, r1
    ret
.main:
    mov r0, 3
    mov r1, 5
    call .func
    add r3, r2, 10
```

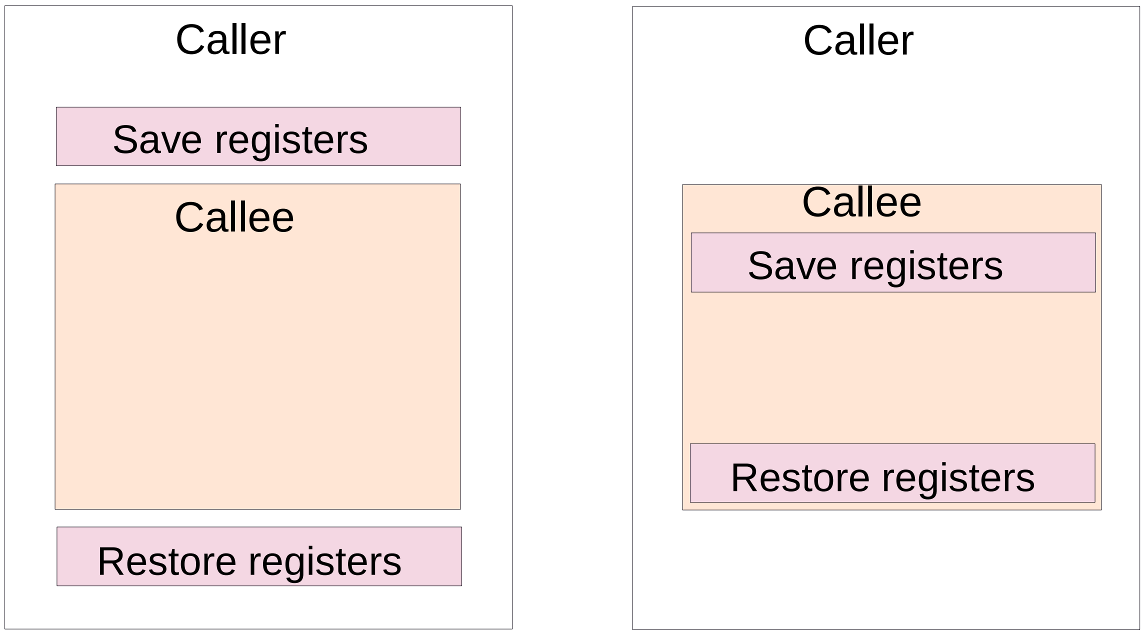
Problems with this Mechanism

* Space Problem
  * We have a limited number of registers
  * We cannot pass more than 16 arguments
  * Solution : Use memory also
* Overwrite Problem
  * What if a function calls itself ? (recursive call)
  * The callee can overwrite the registers of the caller
  * Solution : Spilling

Register Spilling

* The notion of spilling
  * The caller can save the set of registers its needs
  * Call the function
  * And then restore the set of registers after the function returns
  * Known as the caller saved scheme
* callee saved scheme
  * The callee saves the registers, and later restores them

Spilling



(a) Caller saved

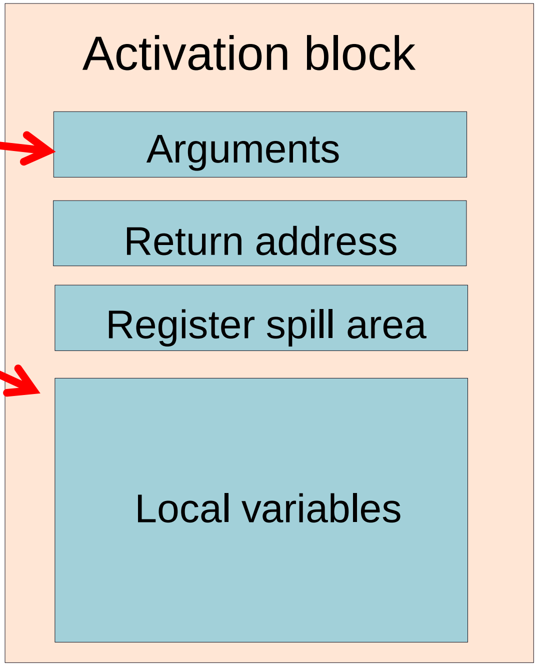
(b) Callee saved

Problems with our Approach

* Using memory, and spilling
  solves both the space problem and overwrite problem
* However, there needs to be :
  * a strict agreement between the caller and the callee regarding the set of memory locations that need to be used
  * Secondly, after a function has finished execution, all the space that it uses needs to be reclaimed

Activation Block

```
int foo(int arg1) {
        int a, b, c;
        a = 3; b = 4;
        c = a + b + arg1;
        return c;
}
```



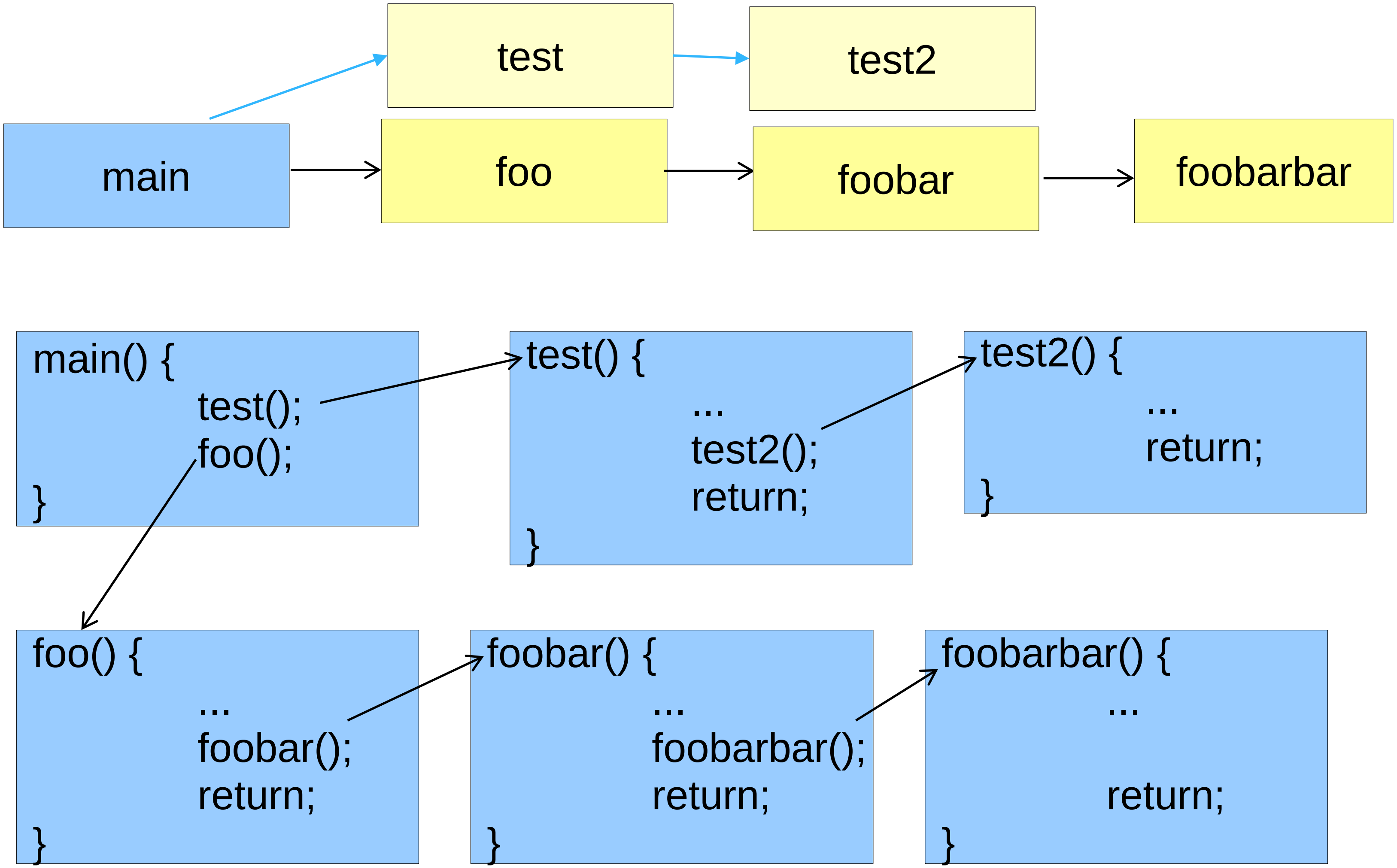
* Activation block → memory map of a function

  arguments, register spill area, local variables

Organising Activation Blocks

* All the information of an executing function is stored in its activation block
* These blocks need to be dynamically created and destroyed – millions of times
* What is the correct way of managing them, and ensuring their fast creation and deletion ?
* Is there a pattern ?

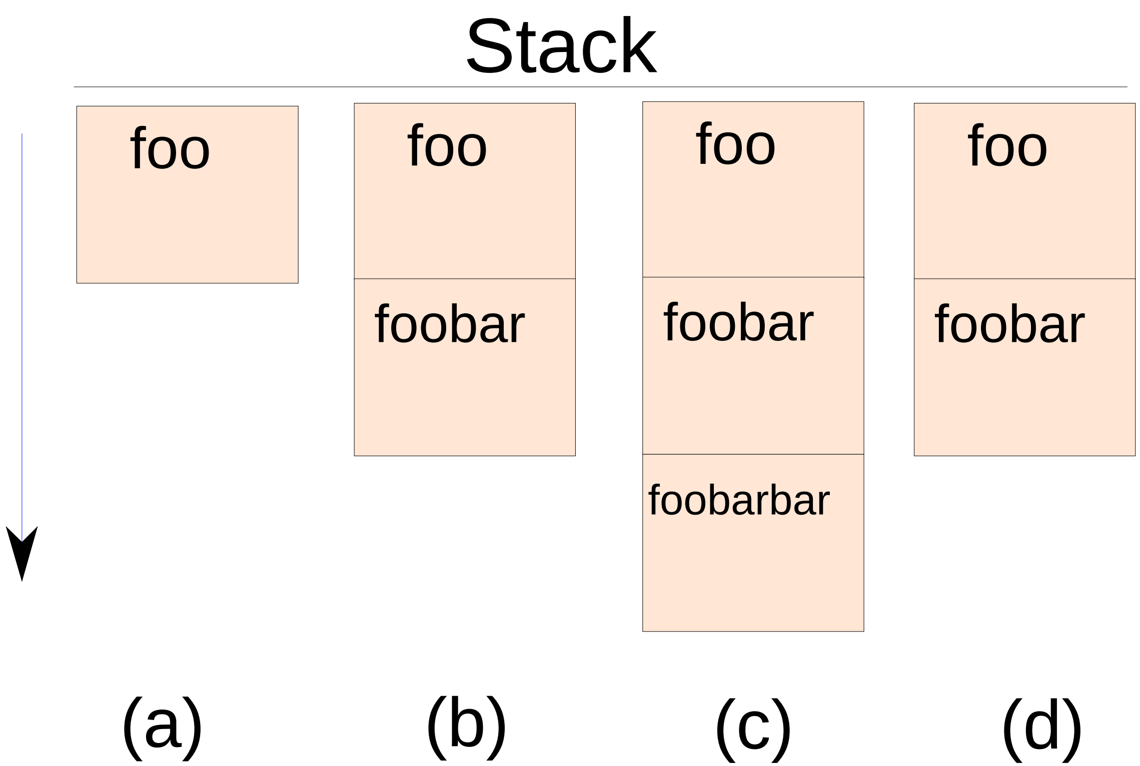
Pattern of Function Calls



Pattern of Function Calls

- Last in First Out

Use a stack to store activation blocks



Working with the Stack

- Allocate a part of the memory to save the stack
- Traditionally stacks are downward growing.
  - The first activation block starts at the highest address
  - Subsequent activation blocks are allocated lower addresses
- The stack pointer register (sp (14)) points to the beginning of an activation block
- Allocating an activation block : sp ← sp - <constant>
- De-allocating an activation block: sp ← sp + <constant>

Issues solved by stack

* Space problem
  * Pass as many parameters as required in the activation block
* Overwrite problem
  * Solved by activation blocks
* Management of activation blocks
  * Solved by the notion of the stack
* The stack needs to primarily be managed in software

call and ret instructions

call .label	$ra \leftarrow PC + 4\ ;\ PC \leftarrow address(.label);$
ret	$PC \leftarrow ra$

- ra (or r15) ← return address register
- call instruction
  - Puts *pc + 4* in ra, and jumps to the function
- ret instruction
  - Puts *ra* in *pc*

Recursive Factorial Program

```
int factorial(int num) {
  if (num <= 1) return 1;
  return num * factorial(num - 1);
}
void main() {
  int result = factorial(10);
}
```

```
.factorial:
   cmp r0, 1        /* compare (1,num) */
   beq .return
   bgt .continue
   b .return
.continue:
   sub sp, sp, 8       /* create space on the stack */

   st r0, [sp]              /* push r0 on the stack */
   st ra, 4[sp]             /* push the return address register */
   sub r0, r0, 1       /* num = num - 1 */
   call .factorial     /* result will be in r1 */
   ld r0, [sp]              /* pop r0 from the stack */
   ld ra, 4[sp]             /* restore the return address */
   mul r1, r0, r1      /* factorial(n) = n * factorial(n-1) */
   add sp, sp, 8       /* delete the activation block */
   ret
.return:
   mov r1, 1
   ret
.main:
   mov r0, 10
   call .factorial
```