CS301 Computer Architecture

DR GAYATHRI ANANTHANARAYANAN

gayathri@iitdh.ac.in

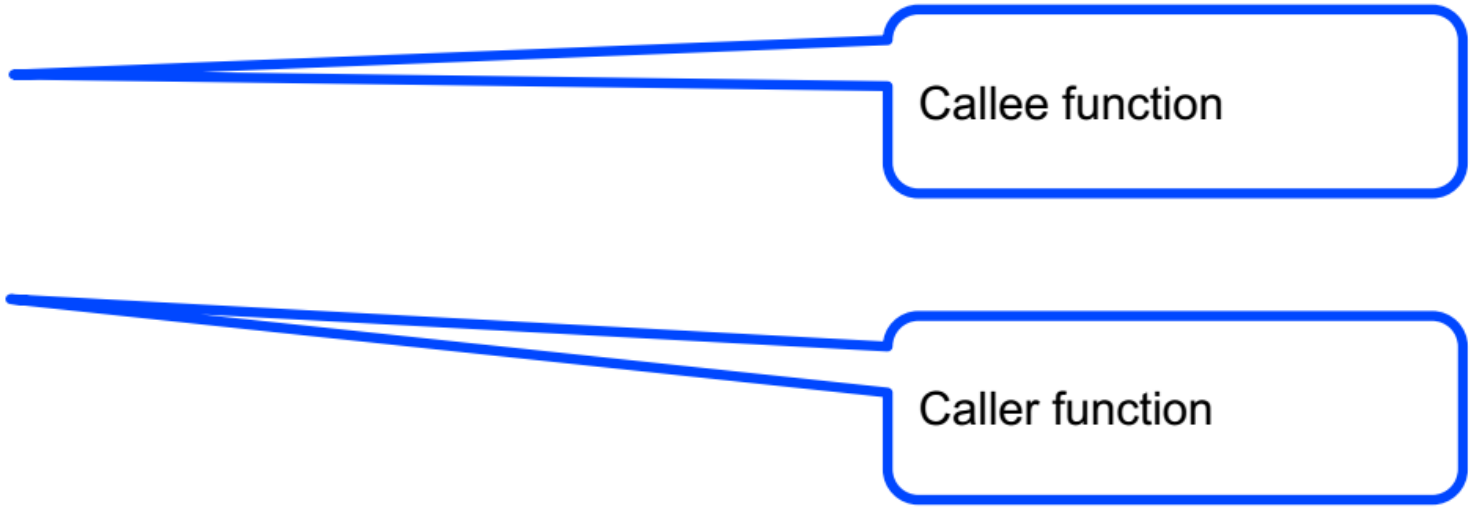
Materials in these slides are borrowed from textbooks and existing Architecture courses

Functions

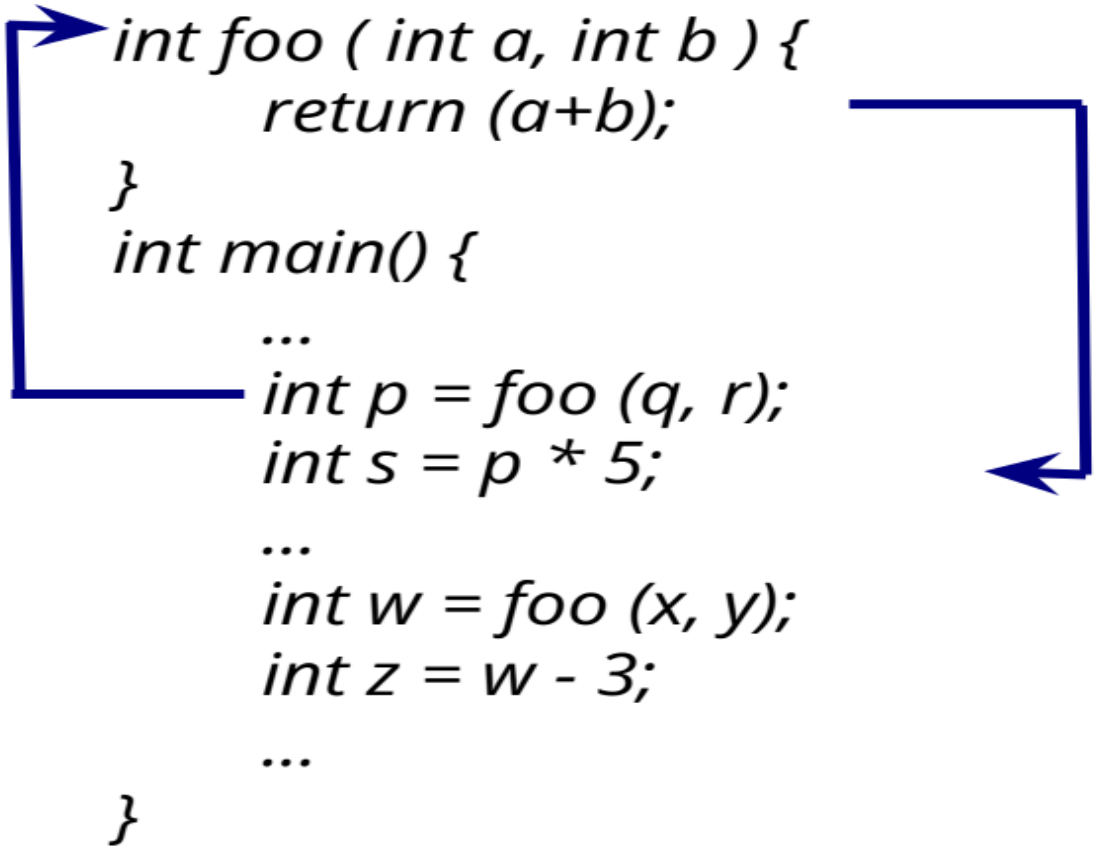
- Instruction sequence that is used frequently
- Typically invoked in more than one context
  - Can even invoke itself
- Notion of input arguments
- Notion of return values
- Notion of state

Example

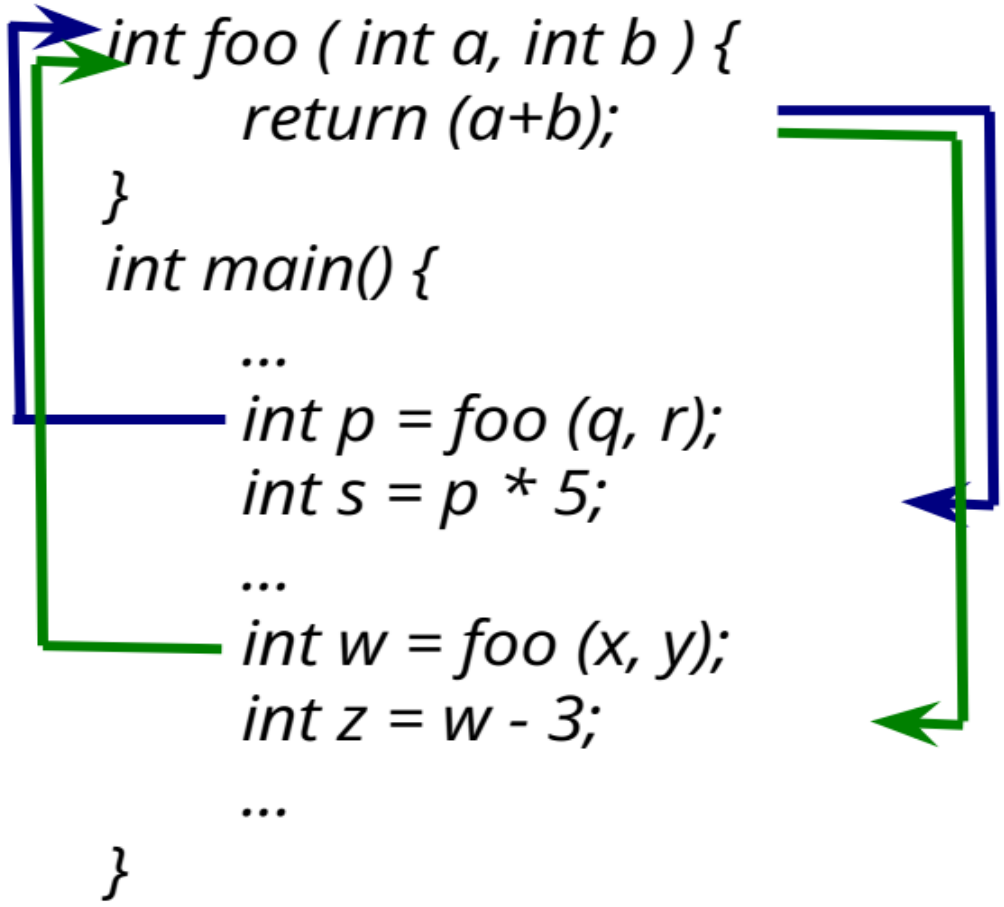
```
int foo ( int a, int b ) {
        return (a+b);
}
int main() {
        …
        int p = foo (q, r);
        int s = p * 5;
        …
        int w = foo (x, y);
        int z = w - 3;
        …
}
```



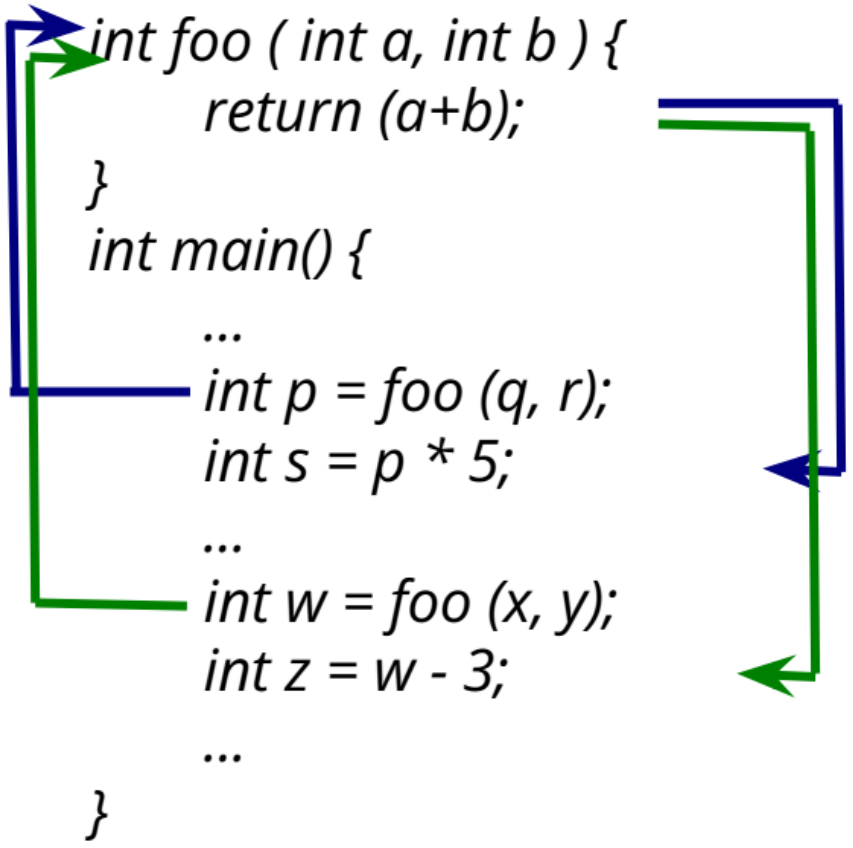
Example


int foo (int a, int b) {
return (a+b);
}
int main() {
...
int p = foo (q, r);
int s = p * 5;
...
int w = foo (x, y);
int z = w - 3;
...
}

Example


int foo (int a, int b) {
return (a+b);
}
int main() {
...
int p = foo (q, r);
int s = p * 5;
...
int w = foo (x, y);
int z = w - 3;
...
}

Control Flow

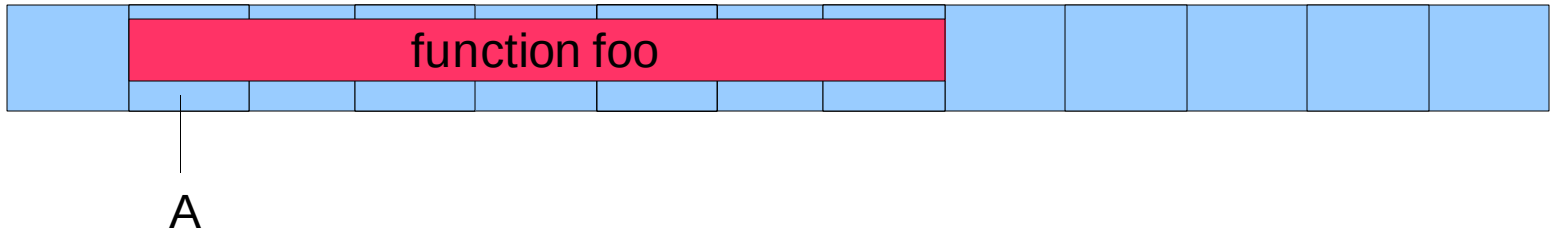



```
.foo:
0x1234  ...
0x1238  ret
...
.main:
0x2000  ...
...
0x2040  call .foo
0x2044  ...
...
0x2100  call .foo
0x2104  ...
...
```

PC	ra
0x2040	

Implementing Functions

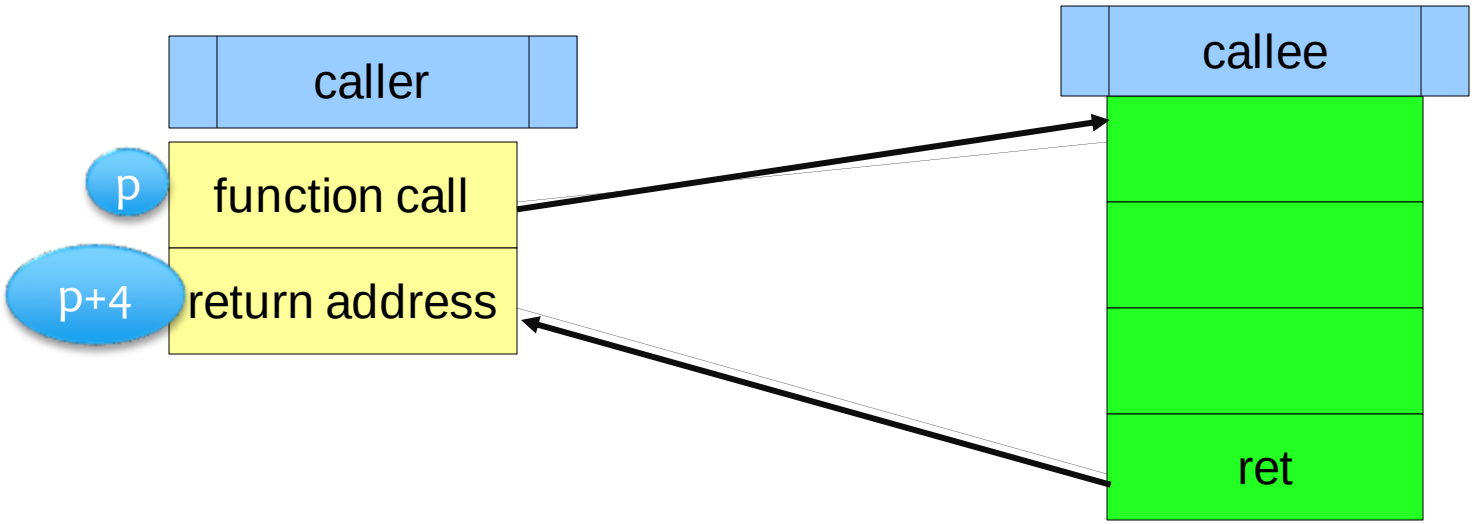
* Functions are blocks of assembly instructions that can be repeatedly invoked to perform a certain action
* Every function has a starting address in memory (e.g. foo has a starting address A



Implementing Functions - II

* To call a function, we need to set :
  * pc ← 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

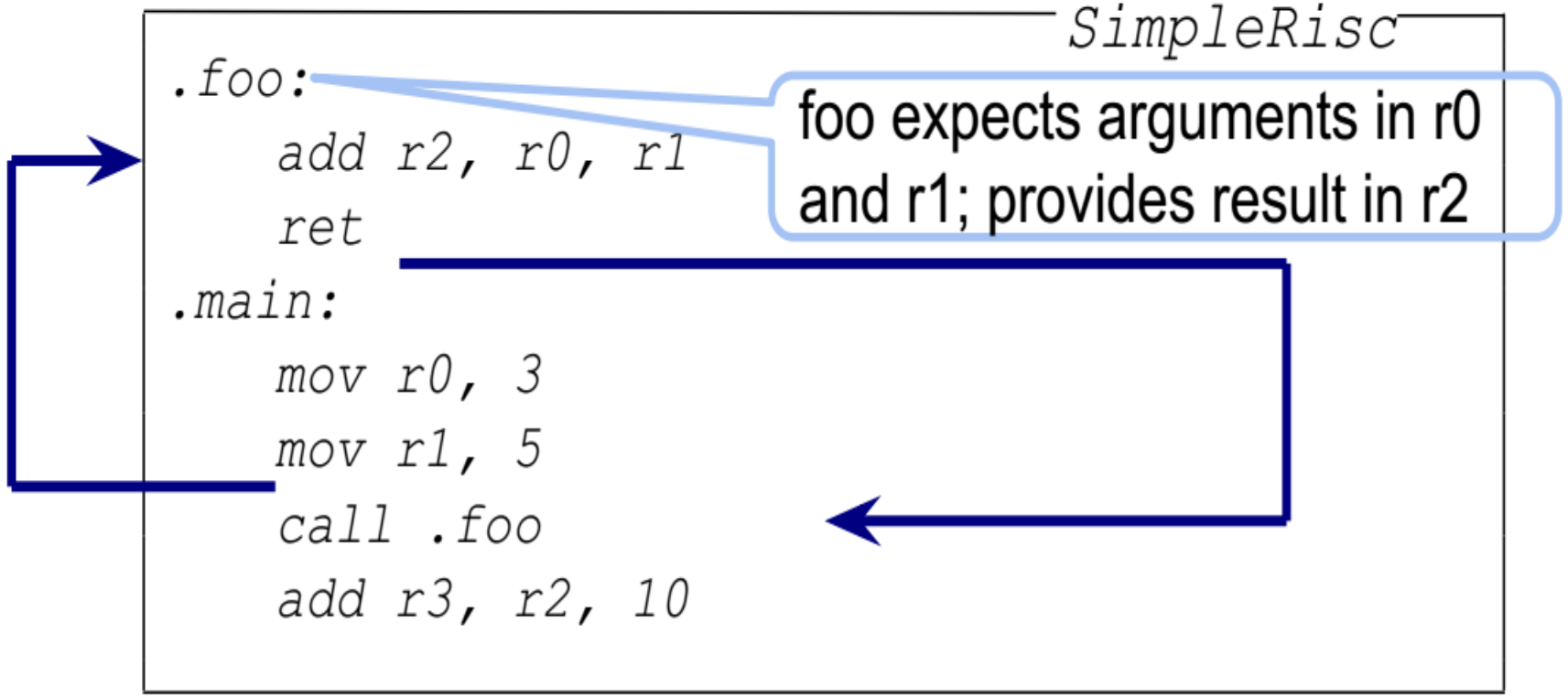
Two New Instructions in the ISA

1. call .foo
   a. ra ← PC + 4
   b. PC ← address(.foo)
2. ret
   a. PC ← ra

Remember:
"ra" or the Return Address register is the same as r15

How do we pass arguments/ return values

* Solution : use registers



Problems with this Mechanism

* Space Problem
  * We have a limited number of registers
  * We cannot pass more than 16 arguments

```
int factorial ( int n ) {
        if ( n == 1 )
                return n;
        else
                return n * factorial ( n - 1 );
}
int main() {
        factorial ( 3 );
}
```

```
.factorial:
0x1000      cmp r2, 1
0x1004      beq .done
0x1008      mov r3, r2
0x100c      sub r2, r2, 1
0x1010      call .factorial
0x1014      mul r1, r1, r3
0x1018      ret
.done:
0x101c      mov r1, 1
0x1020      ret
...
.main:
0x2000      mov r2, 3
0x2004      call .factorial
```

factorial expects argument in r2; provides result in r1

```
int factorial ( int n ) {
        if ( n == 1 )
                return n;
        else
                return n * factorial ( n - 1 );
}
int main() {
        factorial ( 3 );
}
```

Value of r3, which constituted the "state" of the particular invocation/ call of the function, is getting lost.

```
.factorial:
0x1000      cmp r2, 1
0x1004      beq .done
0x1008      mov r3, r2
0x100c      sub r2, r2, 1
0x1010      call .factorial
0x1014      mul r1, r1, r3
0x1018      ret
.done:
0x101c      mov r1, 1
0x1020      ret
...
.main:
0x2000      mov r2, 3
0x2004      call .factorial
```

factorial expects argument in r2; provides result in r1

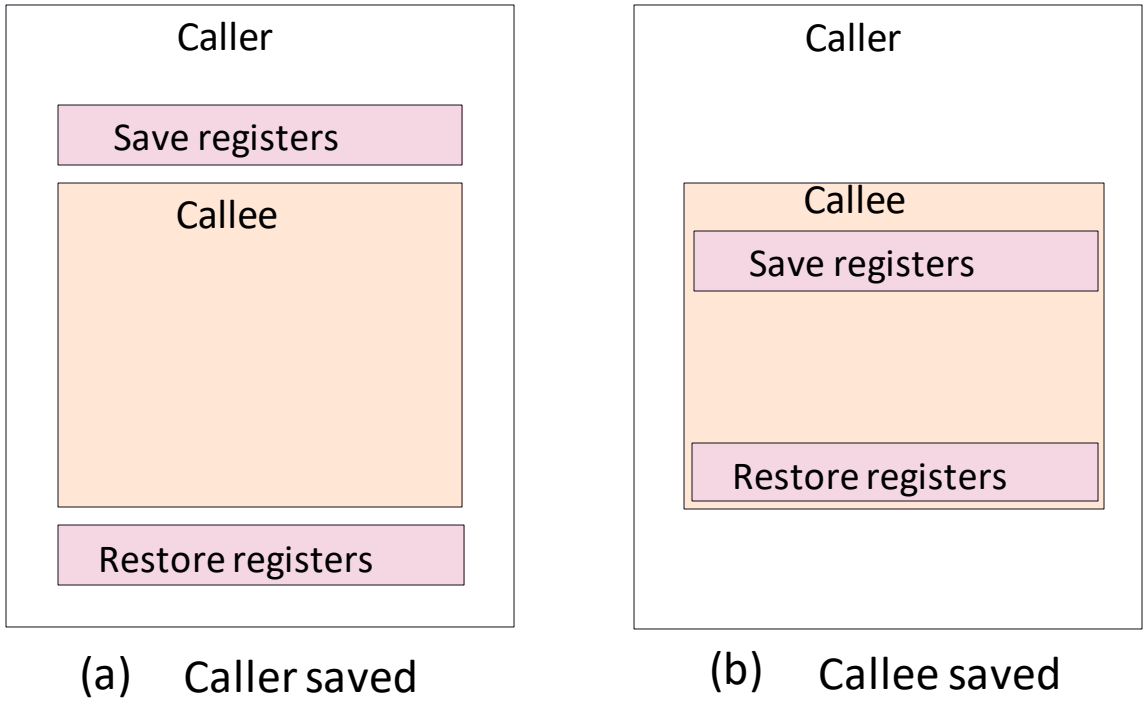
Problems with this Mechanism

* 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



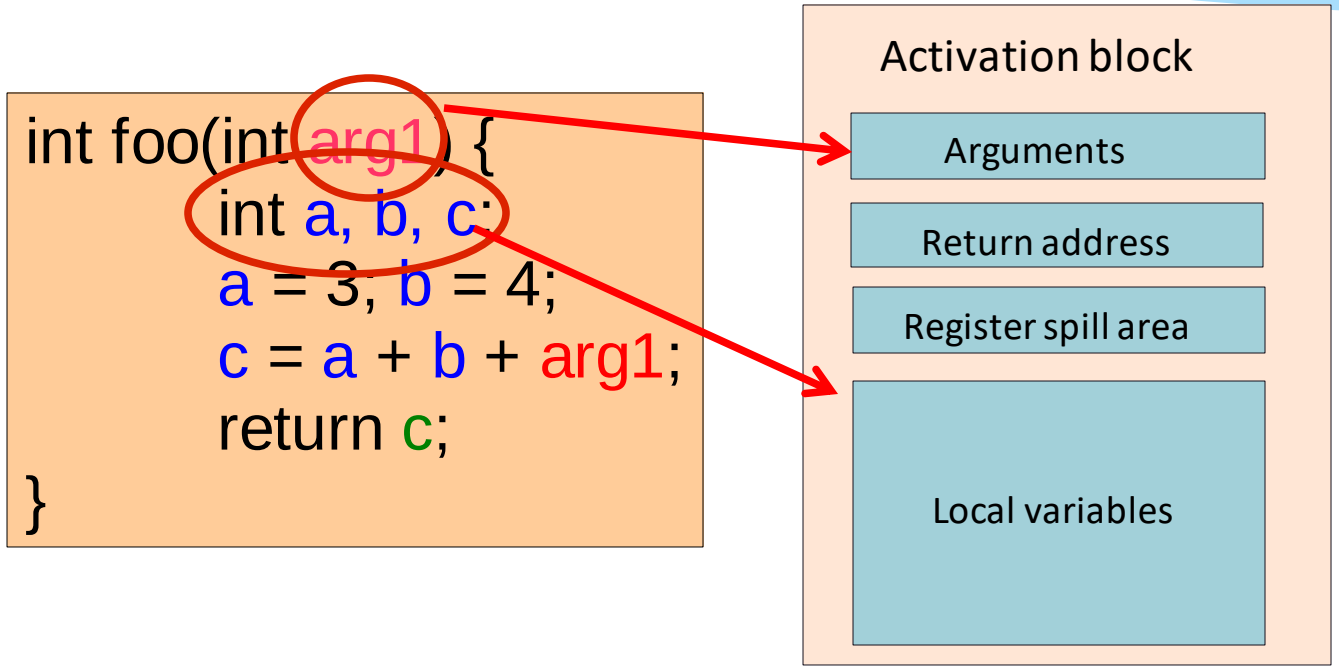
(a) Caller saved

(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



* Activation block → memory map of a function

  arguments, register spill area, local vars

How to use activation blocks ?

* Assume caller saved spilling
* Before calling a function : spill the registers
* Allocate the activation block of the callee
* Write the arguments to the activation block of the callee, if they do not fit in registers
* Call the function

Using Activation Blocks - II

* In the called function
  * Read the arguments and transfer to registers (if required)
  * Save the return address if the called function can call other functions
  * Allocate space for local variables
  * Execute the function
* Once the function ends
  * Restore the value of the return address register (if required)
  * Write the return values to registers, or the activation block of the caller
  * Destroy the activation block of the callee

Using Activation Blocks - III

* Once the function ends (contd …)
  * Call the ret instruction
  * and return to the caller
* The caller :
  * Retrieve the return values from the registers of from its activation block
  * Restore the spilled registers
  * continue …