

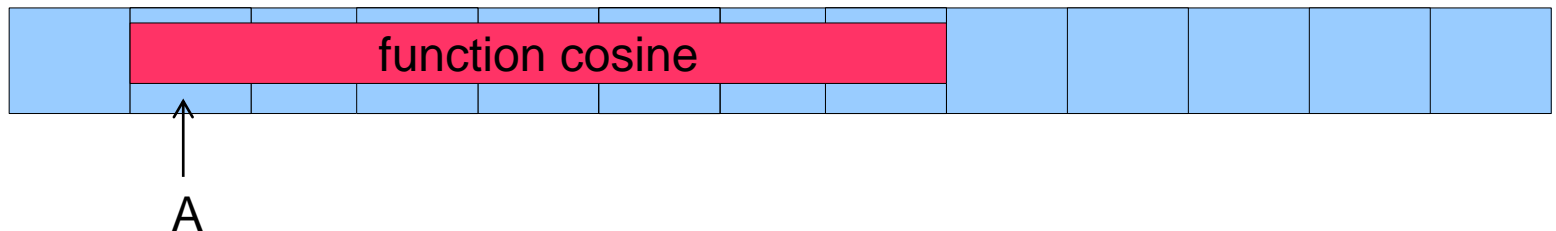
Functions and Stacks

**Slide courtesy: Smruti
Ranjan Sarangi**

Slides adapted by: Dr Sparsh Mittal

Implementing Functions

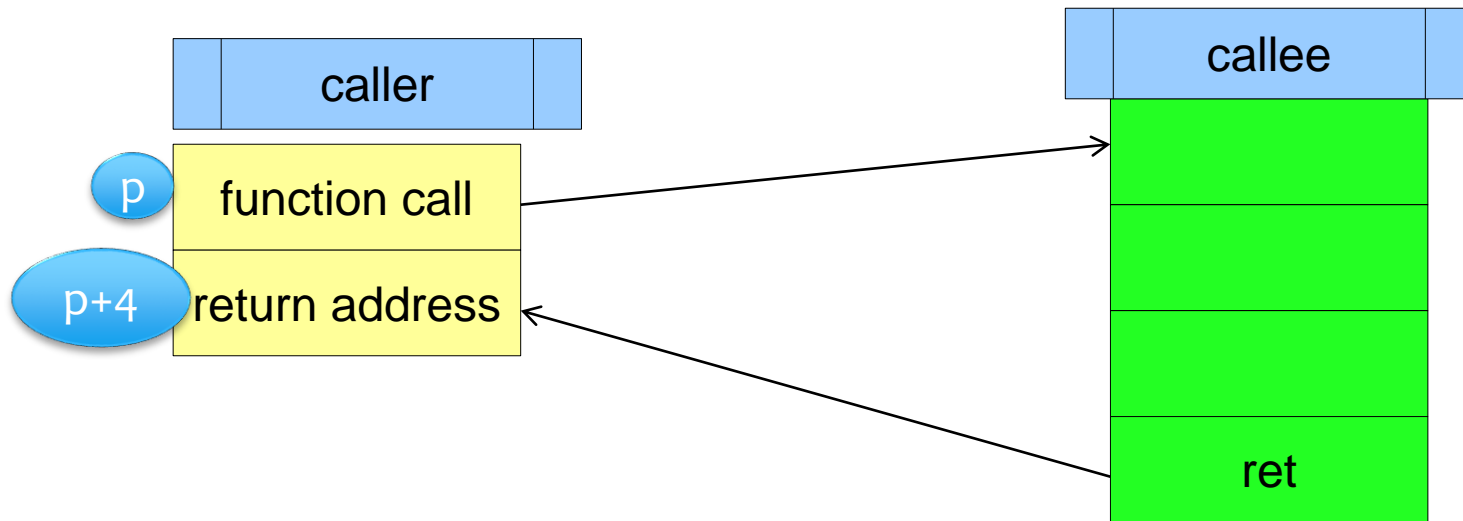
- * Functions are blocks of assembly instructions that can be repeatedly invoked to perform a certain action
- * Every function has a starting address in memory (e.g. cosine has a starting address A)



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



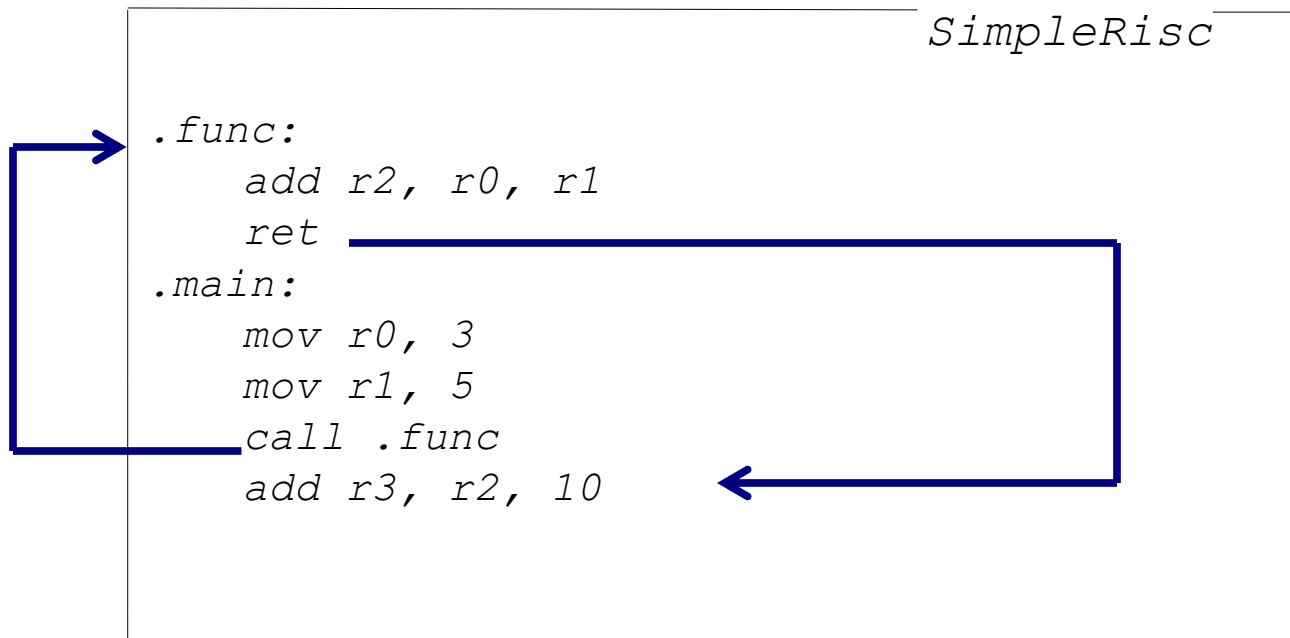
* PC of the call instruction $\rightarrow p$

* PC of the return address $\rightarrow p + 4$

because, every instruction takes 4 bytes

How to 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
- * **Solution** : Use memory also

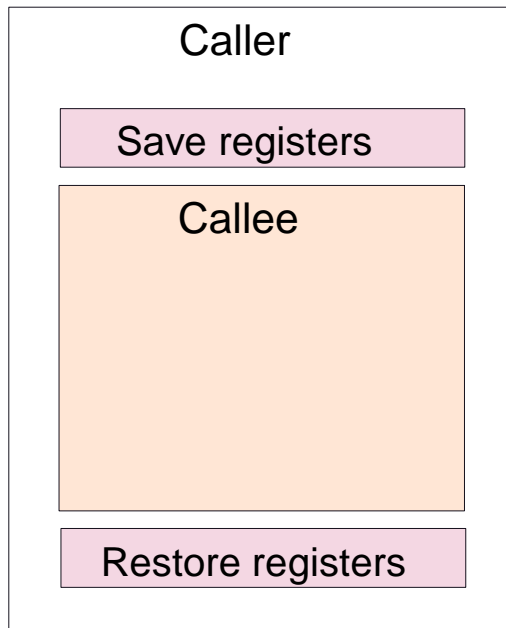
* Overwrite Problem

- * What if a function calls itself ? (recursive call)
- * The callee can **overwrite** the registers of the caller
- * **Solution** : Spilling

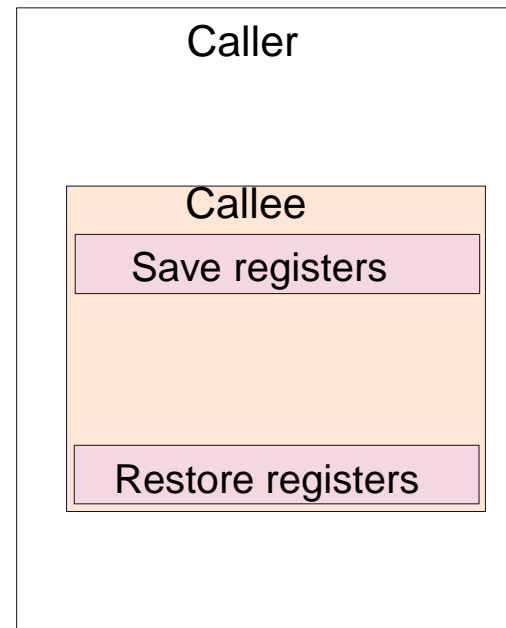
Register Spilling

- * 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

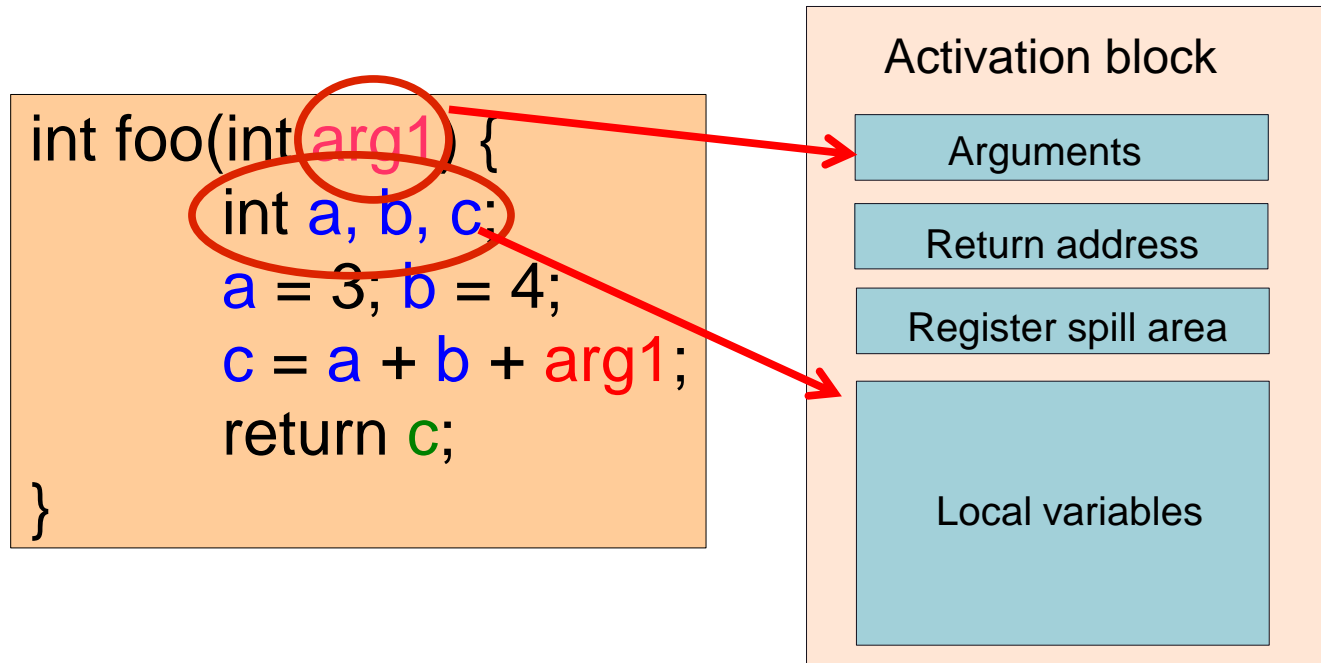


(b) Callee saved

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

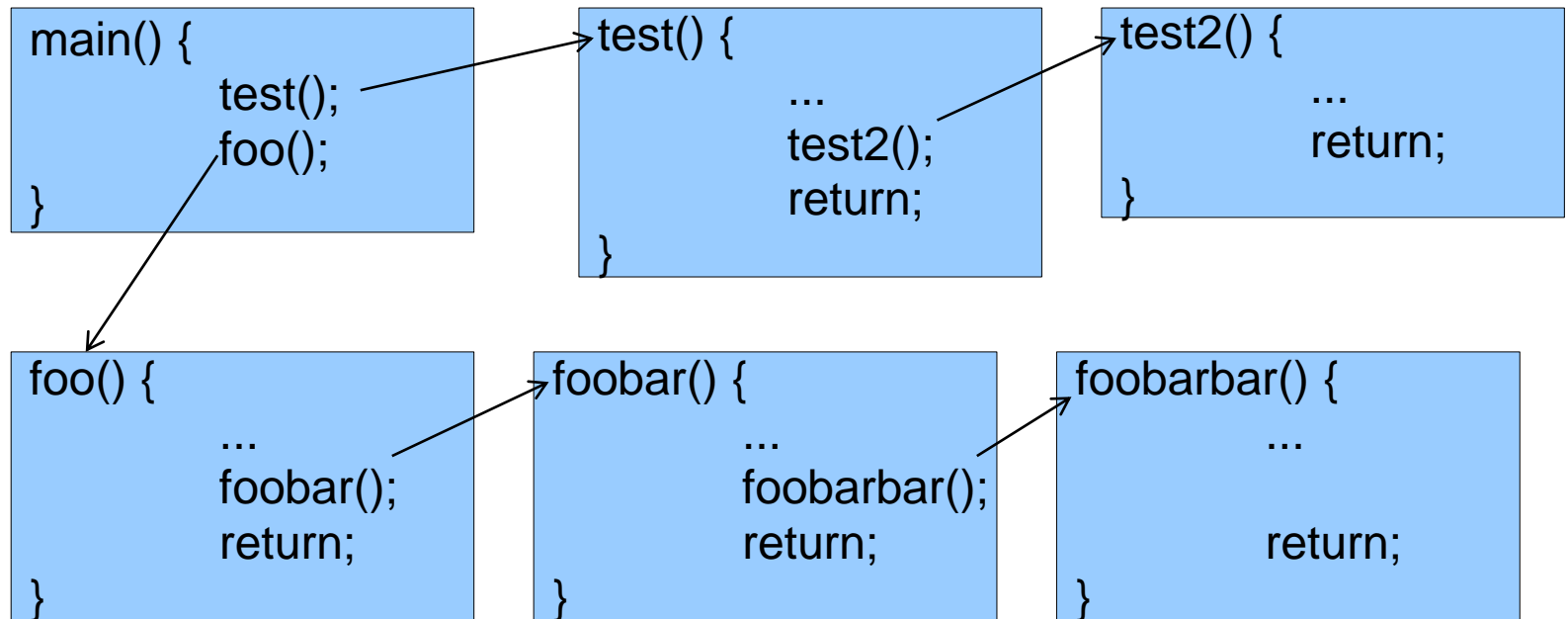
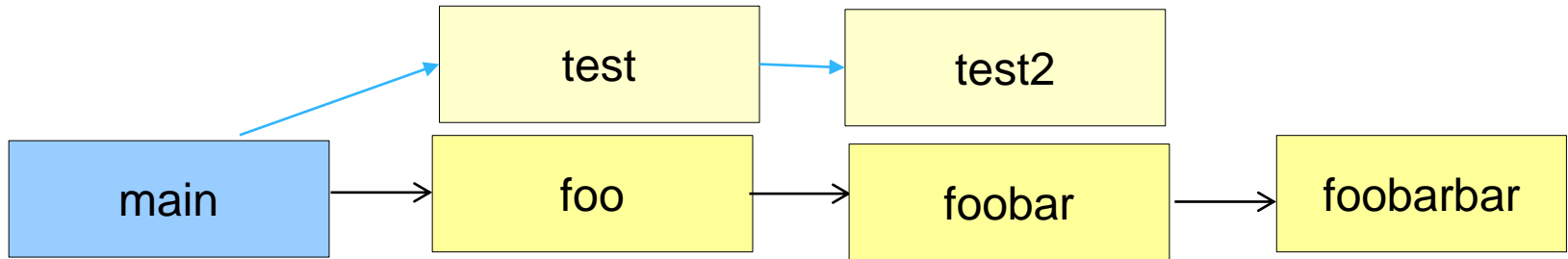


- * **Activation block** → memory map of a function
arguments, register spill area, local variables

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 ?

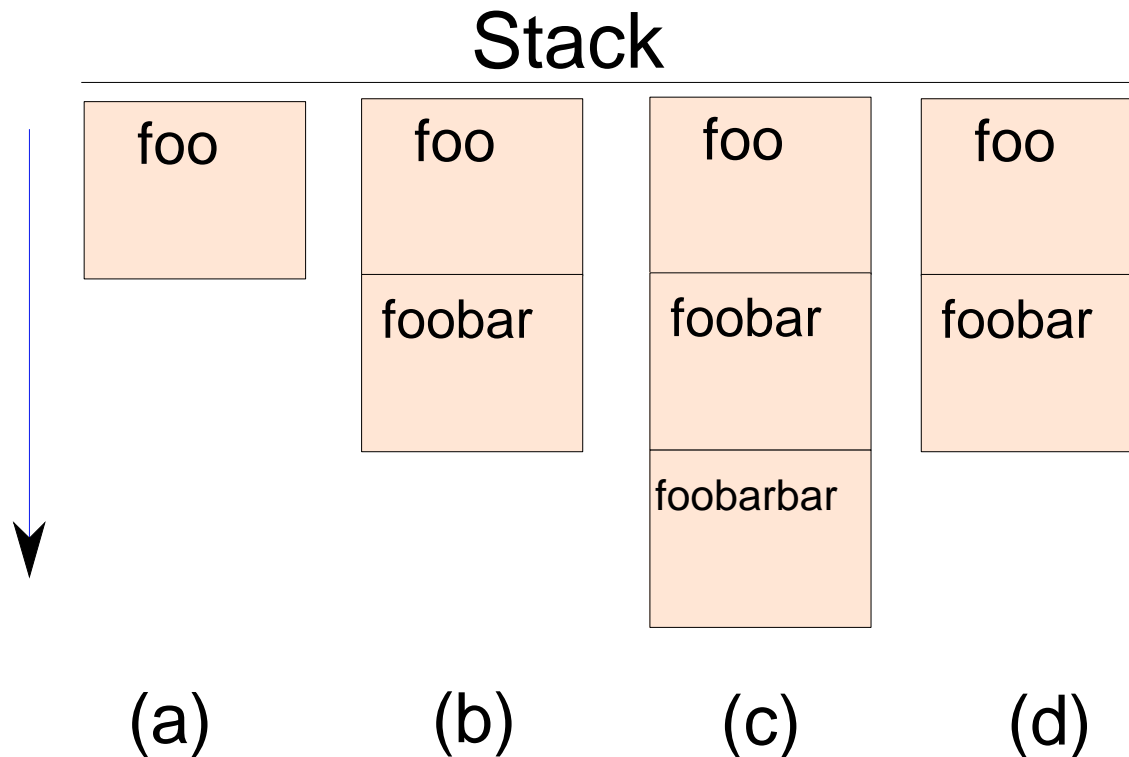
Pattern of Function Calls



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 : $sp \leftarrow sp - \langle \text{constant} \rangle$
- * De-allocating an activation block: $sp \leftarrow sp + \langle \text{constant} \rangle$

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) \leftarrow 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 .initR1
    bgt .continue
    b .initR1

.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

.initR1:
    mov r1, 1
    ret

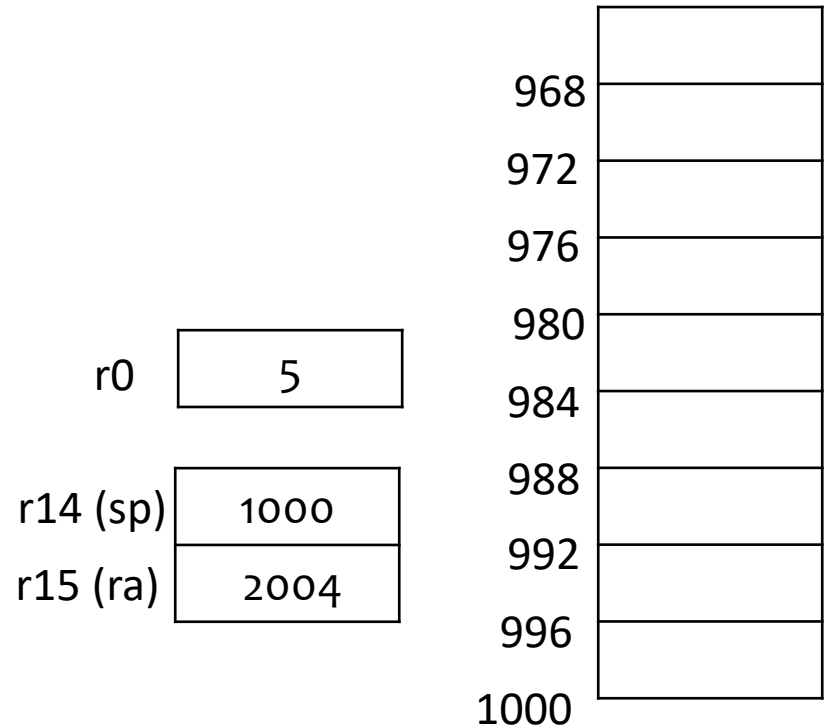
.main:
    mov r0, 5 
    call .factorial /*addr=2000*/

```

```

.factorial:
    cmp r0, 1
    beq .initR1
    bgt .continue
    b .initR1
.continue:
    sub sp, sp, 8 ←
    st r0, [sp]
    st ra, 4[sp]
    sub r0, r0, 1
    call .factorial
    ld r0, [sp]
    ld ra, 4[sp]
    mul r1, r0, r1
    add sp, sp, 8
    ret
.initR1:
    mov r1, 1
    ret
.main:
    mov r0, 5
    call .factorial

```

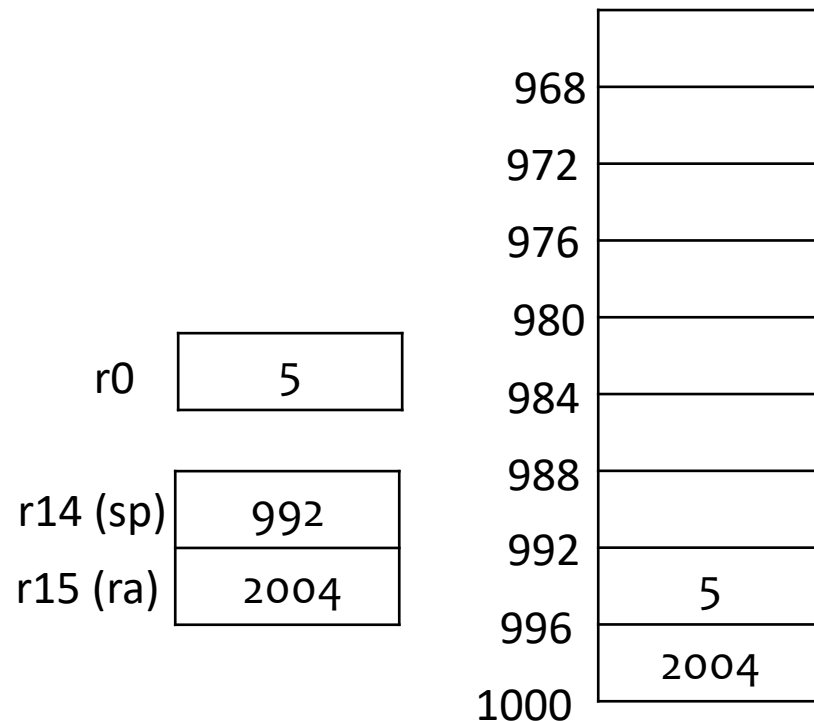


```

.factorial:
    cmp r0, 1
    beq .initR1
    bgt .continue
    b .initR1
.continue:
    sub sp, sp, 8

    st r0, [sp]
    st ra, 4[sp]
    sub r0, r0, 1 ←
    call .factorial
    ld r0, [sp]
    ld ra, 4[sp]
    mul r1, r0, r1
    add sp, sp, 8
    ret
.initR1:
    mov r1, 1
    ret
.main:
    mov r0, 5
    call .factorial

```



```

.factorial:
    cmp r0, 1
    beq .initR1
    bgt .continue
    b .initR1

```



```

.continue:
    sub sp, sp, 8

```

```

    st r0, [sp]
    st ra, 4[sp]
    sub r0, r0, 1
    call .factorial /*assume address =3000 */
    ld r0, [sp]
    ld ra, 4[sp]
    mul r1, r0, r1
    add sp, sp, 8
    ret

```

```

.initR1:
    mov r1, 1
    ret

```

```

.main:
    mov r0, 5
    call .factorial

```

r0	4
r14 (sp)	992
r15 (ra)	3004

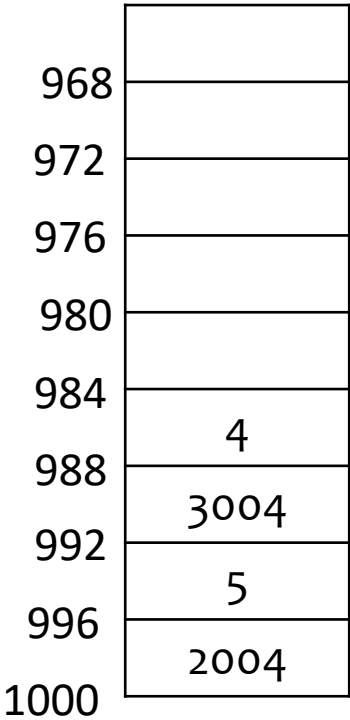
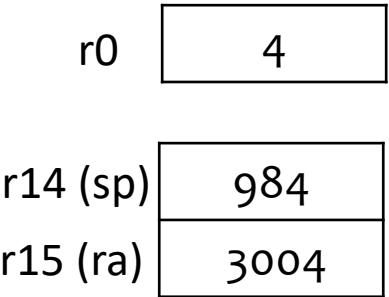
968	
972	
976	
980	
984	
988	
992	5
996	2004
1000	

```

.factorial:
    cmp r0, 1
    beq .initR1
    bgt .continue
    b .initR1
.continue:
    sub sp, sp, 8

    st r0, [sp]
    st ra, 4[sp]
    sub r0, r0, 1
    call .factorial /*assume address =3000 */
    ld r0, [sp]
    ld ra, 4[sp]
    mul r1, r0, r1
    add sp, sp, 8
    ret
.initR1:
    mov r1, 1
    ret
.main:
    mov r0, 5
    call .factorial

```



```

.factorial:
    cmp r0, 1
    beq .initR1
    bgt .continue
    b .initR1

```



```

.continue:
    sub sp, sp, 8

```

```

    st r0, [sp]
    st ra, 4[sp]
    sub r0, r0, 1
    call .factorial /*assume address =3000 */
    ld r0, [sp]
    ld ra, 4[sp]
    mul r1, r0, r1
    add sp, sp, 8
    ret

```

```

.initR1:
    mov r1, 1
    ret

```

```

.main:
    mov r0, 5
    call .factorial

```

r0	3
r14 (sp)	984
r15 (ra)	3004

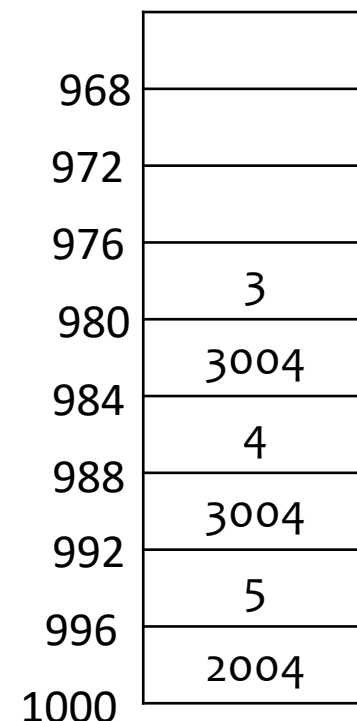
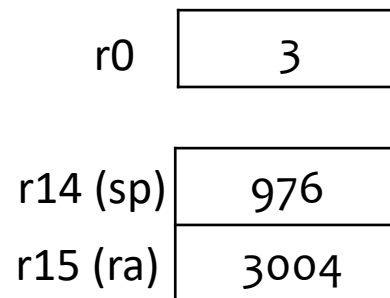
968	
972	
976	
980	
984	4
988	3004
992	5
996	2004
1000	

```

.factorial:
    cmp r0, 1
    beq .initR1
    bgt .continue
    b .initR1
.continue:
    sub sp, sp, 8

    st r0, [sp]
    st ra, 4[sp]
    sub r0, r0, 1
    call .factorial /*assume address =3000 */
    ld r0, [sp]
    ld ra, 4[sp]
    mul r1, r0, r1
    add sp, sp, 8
    ret
.initR1:
    mov r1, 1
    ret
.main:
    mov r0, 5
    call .factorial

```




```

.factorial:
    cmp r0, 1
    beq .initR1
    bgt .continue
    b .initR1

```



```

.continue:
    sub sp, sp, 8

```

```

    st r0, [sp]
    st ra, 4[sp]
    sub r0, r0, 1
    call .factorial /*assume address =3000 */
    ld r0, [sp]
    ld ra, 4[sp]
    mul r1, r0, r1
    add sp, sp, 8
    ret

```

```

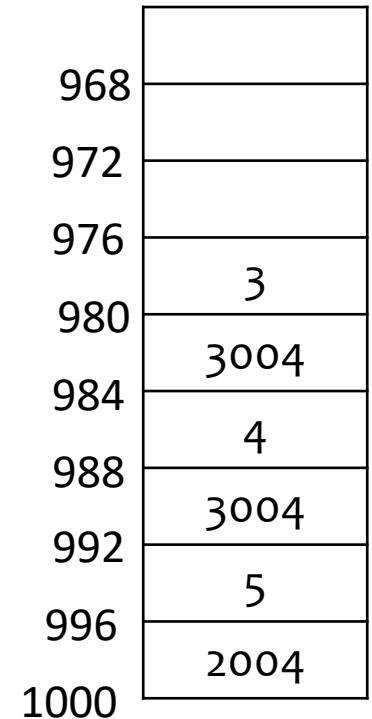
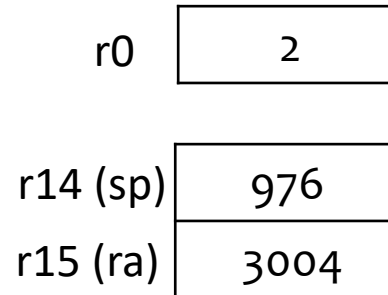
.initR1:
    mov r1, 1
    ret

```

```

.main:
    mov r0, 5
    call .factorial

```

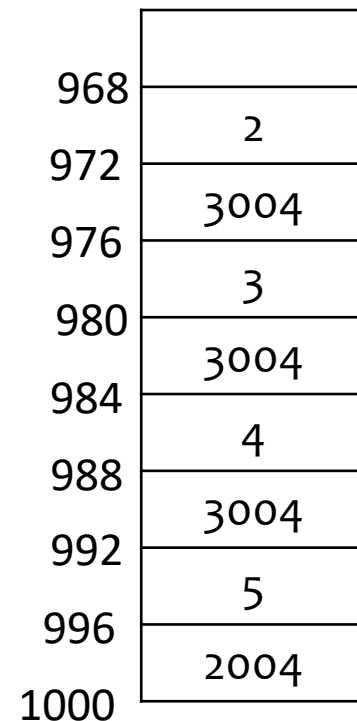
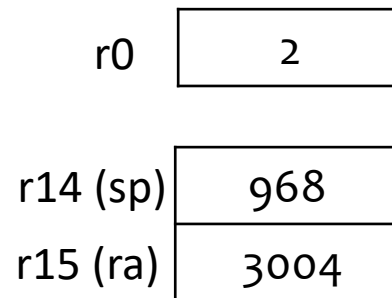


```

.factorial:
    cmp r0, 1
    beq .initR1
    bgt .continue
    b .initR1
.continue:
    sub sp, sp, 8

    st r0, [sp]
    st ra, 4[sp]
    sub r0, r0, 1
    call .factorial /*assume address =3000 */
    ld r0, [sp]
    ld ra, 4[sp]
    mul r1, r0, r1
    add sp, sp, 8
    ret
.initR1:
    mov r1, 1
    ret
.main:
    mov r0, 5
    call .factorial

```



```

.factorial:
    cmp r0, 1
    beq .initR1
    bgt .continue
    b .initR1

.continue:
    sub sp, sp, 8

    st r0, [sp]
    st ra, 4[sp]
    sub r0, r0, 1
    call .factorial /*assume address =3000 */
    ld r0, [sp]
    ld ra, 4[sp]
    mul r1, r0, r1
    add sp, sp, 8
    ret

.initR1:
    mov r1, 1
    ret

.main:
    mov r0, 5
    call .factorial

```



r0	1
r1	
r14 (sp)	968
r15 (ra)	3004

968	
972	2
976	3004
980	3
984	3004
988	4
992	3004
996	5
1000	2004

```

.factorial:
    cmp r0, 1
    beq .initR1
    bgt .continue
    b .initR1

```

```

.continue:
    sub sp, sp, 8

```

```

    st r0, [sp]
    st ra, 4[sp]
    sub r0, r0, 1
    call .factorial /*assume address =3000 */
    ld r0, [sp]
    ld ra, 4[sp]
    mul r1, r0, r1
    add sp, sp, 8
    ret

```

```

.initR1:
    mov r1, 1
    ret /* PC <- ra */

```

```

.main:
    mov r0, 5
    call .factorial

```

r0	1
r1	1
r14 (sp)	968
r15 (ra)	3004

968	
972	2
976	3004
980	3
984	3004
988	4
992	3004
996	5
1000	2004



```

.factorial:
    cmp r0, 1
    beq .initR1
    bgt .continue
    b .initR1

.continue:
    sub sp, sp, 8

    st r0, [sp]
    st ra, 4[sp]
    sub r0, r0, 1
    call .factorial /*assume address =3000 */
    ld r0, [sp]
    ld ra, 4[sp]
    mul r1, r0, r1
    add sp, sp, 8
    ret

.initR1:
    mov r1, 1
    ret /* PC <- ra */

.main:
    mov r0, 5
    call .factorial

```



r0	1
r1	1
r14 (sp)	968
r15 (ra)	3004

968	
972	2
976	3004
980	3
984	3004
988	4
992	3004
996	5
1000	2004

```

.factorial:
    cmp r0, 1
    beq .initR1
    bgt .continue
    b .initR1

.continue:
    sub sp, sp, 8

    st r0, [sp]
    st ra, 4[sp]
    sub r0, r0, 1
    call .factorial /*assume address =3000 */
    ld r0, [sp]
    ld ra, 4[sp]
    mul r1, r0, r1
    add sp, sp, 8
    ret
.initR1:
    mov r1, 1
    ret /* PC <- ra */
.main:
    mov r0, 5
    call .factorial

```



r0	2
r1	2
r14 (sp)	976
r15 (ra)	3004

968	
972	2
976	3004
980	3
984	3004
988	4
992	3004
996	5
1000	2004

```

.factorial:
    cmp r0, 1
    beq .initR1
    bgt .continue
    b .initR1

.continue:
    sub sp, sp, 8

    st r0, [sp]
    st ra, 4[sp]
    sub r0, r0, 1
    call .factorial /*assume address =3000 */
    ld r0, [sp]
    ld ra, 4[sp]
    mul r1, r0, r1
    add sp, sp, 8
    ret

.initR1:
    mov r1, 1
    ret /* PC <- ra */

.main:
    mov r0, 5
    call .factorial

```



r0	2
r1	2
r14 (sp)	976
r15 (ra)	3004

968	
972	2
976	3004
980	3
984	3004
988	4
992	3004
996	5
1000	2004

```

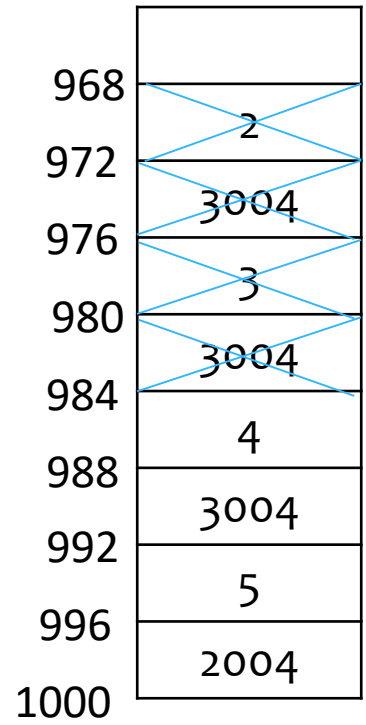
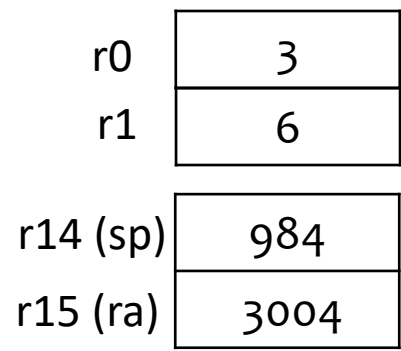
.factorial:
    cmp r0, 1
    beq .initR1
    bgt .continue
    b .initR1

.continue:
    sub sp, sp, 8

    st r0, [sp]
    st ra, 4[sp]
    sub r0, r0, 1
    call .factorial /*assume address =3000 */
    ld r0, [sp]
    ld ra, 4[sp]
    mul r1, r0, r1
    add sp, sp, 8
    ret
    ←
.initR1:
    mov r1, 1
    ret /* PC <- ra */

.main:
    mov r0, 5
    call .factorial

```




```

.factorial:
    cmp r0, 1
    beq .initR1
    bgt .continue
    b .initR1

.continue:
    sub sp, sp, 8

    st r0, [sp]
    st ra, 4[sp]
    sub r0, r0, 1
    call .factorial /*assume address =3000 */
    ld r0, [sp]
    ld ra, 4[sp]
    mul r1, r0, r1
    add sp, sp, 8
    ret

.initR1:
    mov r1, 1
    ret /* PC <- ra */

.main:
    mov r0, 5
    call .factorial

```




r0	3
r1	6
r14 (sp)	984
r15 (ra)	3004

968	
972	2
976	3004
980	3
984	3004
988	4
992	3004
996	5
1000	2004

```

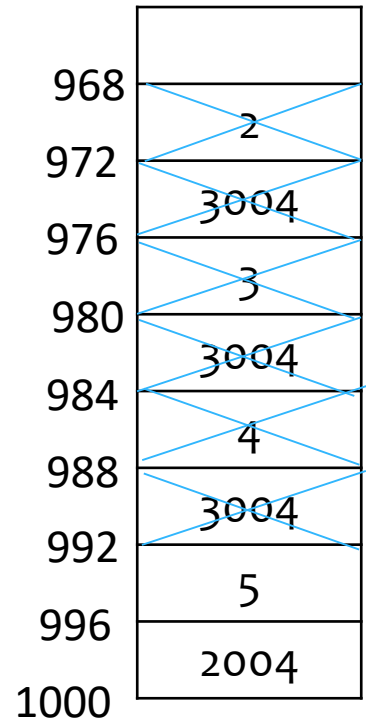
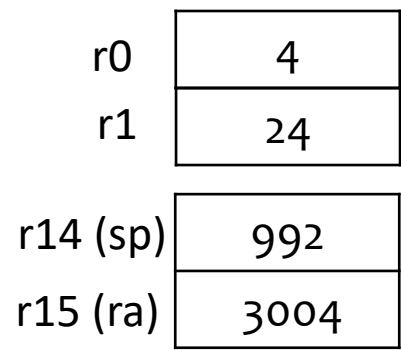
.factorial:
    cmp r0, 1
    beq .initR1
    bgt .continue
    b .initR1

.continue:
    sub sp, sp, 8

    st r0, [sp]
    st ra, 4[sp]
    sub r0, r0, 1
    call .factorial /*assume address =3000 */
    ld r0, [sp]
    ld ra, 4[sp]
    mul r1, r0, r1
    add sp, sp, 8
    ret
    
.initR1:
    mov r1, 1
    ret /* PC <- ra */

.main:
    mov r0, 5
    call .factorial

```



```

.factorial:
    cmp r0, 1
    beq .initR1
    bgt .continue
    b .initR1

.continue:
    sub sp, sp, 8

    st r0, [sp]
    st ra, 4[sp]
    sub r0, r0, 1
    call .factorial /*assume address =3000 */
    ld r0, [sp]
    ld ra, 4[sp]
    mul r1, r0, r1
    add sp, sp, 8
    ret

.initR1:
    mov r1, 1
    ret /* PC <- ra */

.main:
    mov r0, 5
    call .factorial

```



r0	4
r1	24
r14 (sp)	992
r15 (ra)	3004

968	
972	2
976	3004
980	3
984	3004
988	4
992	3004
996	5
1000	2004

```

.factorial:
    cmp r0, 1
    beq .initR1
    bgt .continue
    b .initR1

.continue:
    sub sp, sp, 8

    st r0, [sp]
    st ra, 4[sp]
    sub r0, r0, 1
    call .factorial /*assume address =3000 */
    ld r0, [sp]
    ld ra, 4[sp]
    mul r1, r0, r1
    add sp, sp, 8
    ret
    ←
.initR1:
    mov r1, 1
    ret /* PC <- ra */

.main:
    mov r0, 5
    call .factorial

```

r0	5
r1	120
r14 (sp)	1000
r15 (ra)	2004

968	
972	2
976	3004
980	3
984	3004
988	4
992	3004
996	5
1000	2004

```

.factorial:
    cmp r0, 1
    beq .initR1
    bgt .continue
    b .initR1

.continue:
    sub sp, sp, 8

    st r0, [sp]
    st ra, 4[sp]
    sub r0, r0, 1
    call .factorial /*assume address =3000 */
    ld r0, [sp]
    ld ra, 4[sp]
    mul r1, r0, r1
    add sp, sp, 8
    ret

.initR1:
    mov r1, 1
    ret /* PC <- ra */

.main:
    mov r0, 5
    call .factorial

```

r0	5
r1	120
r14 (sp)	1000
r15 (ra)	2004

968	
972	2
976	3004
980	3
984	3004
988	4
992	3004
996	5
1000	2004

