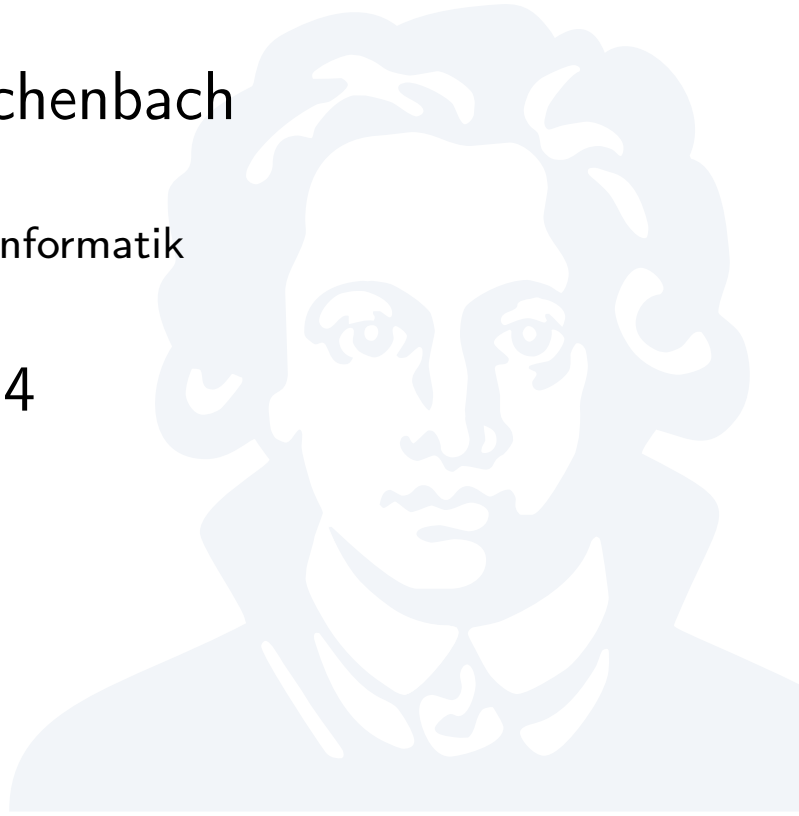
Foundations of Programming Languages

2OPM Assembly (3/3): Subroutines

Prof. Dr. Christoph Reichenbach

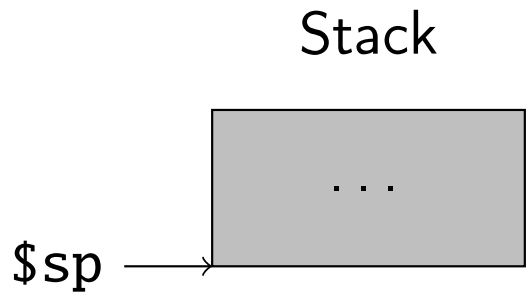
Fachbereich 12 / Institut für Informatik

17. Oktober 2014



Stack Movement

Special operations **push**, **pop** for stack access:

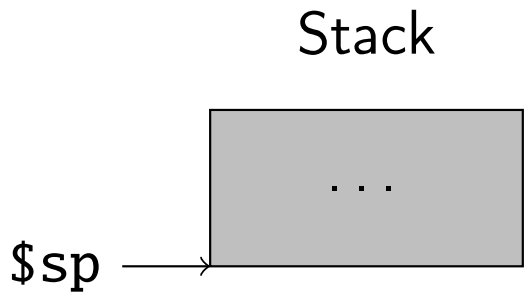



```
⇒   li    $t0, 17
    push  $t0
    push  $t0
    pop   $t1
    ...
```

Processor
$t0 = 00000064
$t1 = 00000000
$sp = 7f...f28

Stack Movement

Special operations **push**, **pop** for stack access:

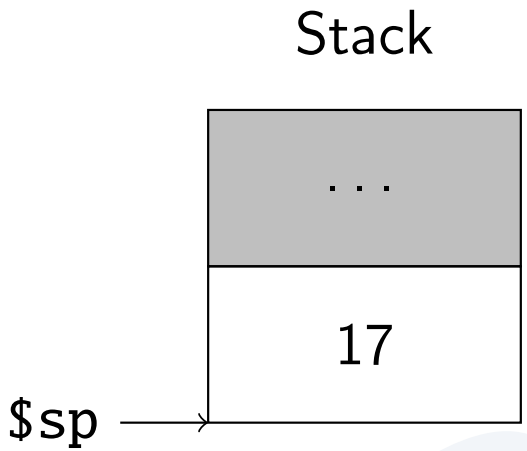



```
    li    $t0, 17
⇒   push  $t0
    push  $t0
    pop   $t1
    ...
```

Processor
$t0 = 00000017
$t1 = 00000000
$sp = 7f...f28

Stack Movement

Special operations **push**, **pop** for stack access:



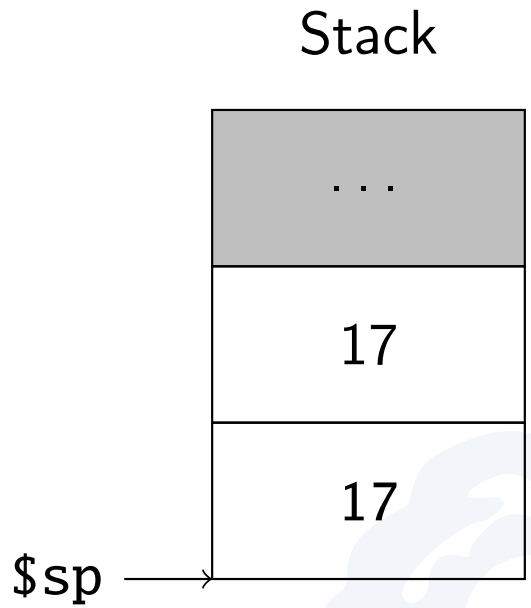

```
    li    $t0, 17
    push  $t0
⇒   push  $t0
    pop   $t1
    ...
```

Processor
$t0 = 00000017
$t1 = 00000000
$sp = 7f...f20

Stack Movement

Special operations **push**, **pop** for stack access:

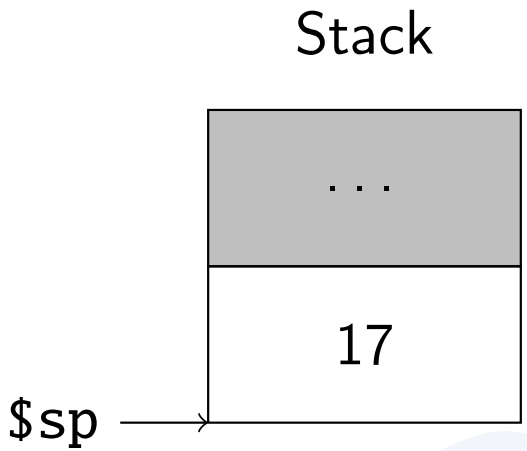
```
    li    $t0, 17
    push  $t0
    push  $t0
⇒   pop   $t1
    ...
```



Processor
$t0 = 00000017
$t1 = 00000000
$sp = 7f...f18

Stack Movement

Special operations **push**, **pop** for stack access:




```
    li    $t0, 17
    push  $t0
    push  $t0
    pop   $t1
⇒   ...
```

Processor
$t0 = 00000017
$t1 = 00000017
$sp = 7f...f20

Stack Movement

Special operations **push**, **pop** for stack access:

Pushing

```
push $r
```

same as:

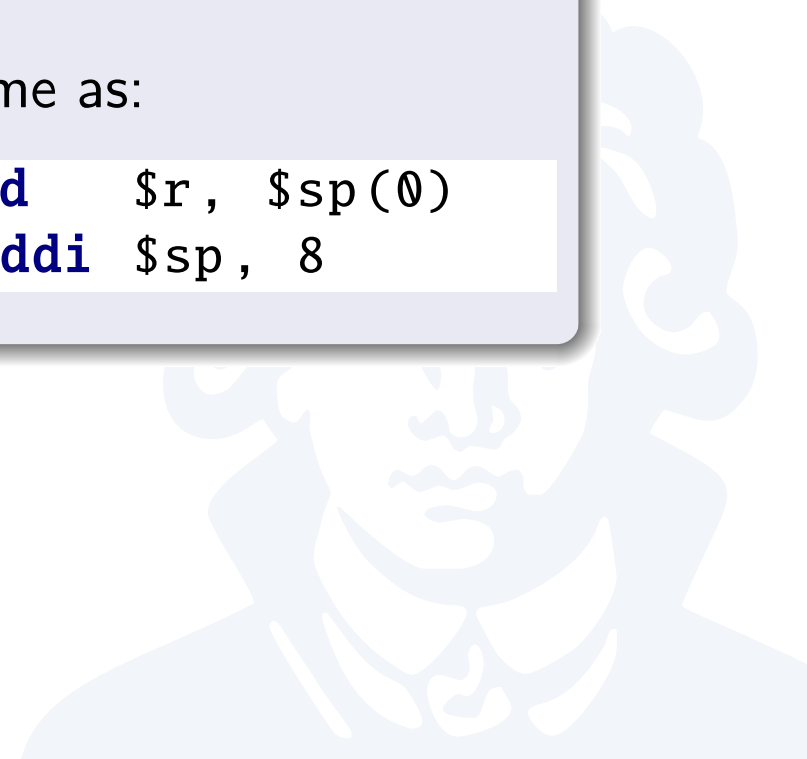
```
subi $sp, 8
sd   $r, $sp(0)
```

Popping

```
pop $r
```

same as:

```
ld   $r, $sp(0)
addi $sp, 8
```



Stack Movement

Special operations **push**, **pop** for stack access:

Pushing

```
push $r
```

same as:

```
subi $sp, 8
sd   $r, $sp(0)
```

Popping

```
pop $r
```

same as:

```
ld   $r, $sp(0)
addi $sp, 8
```

Frequent operations, very compact machine code

Re-use through Subroutines

Let's recall our factorial code:

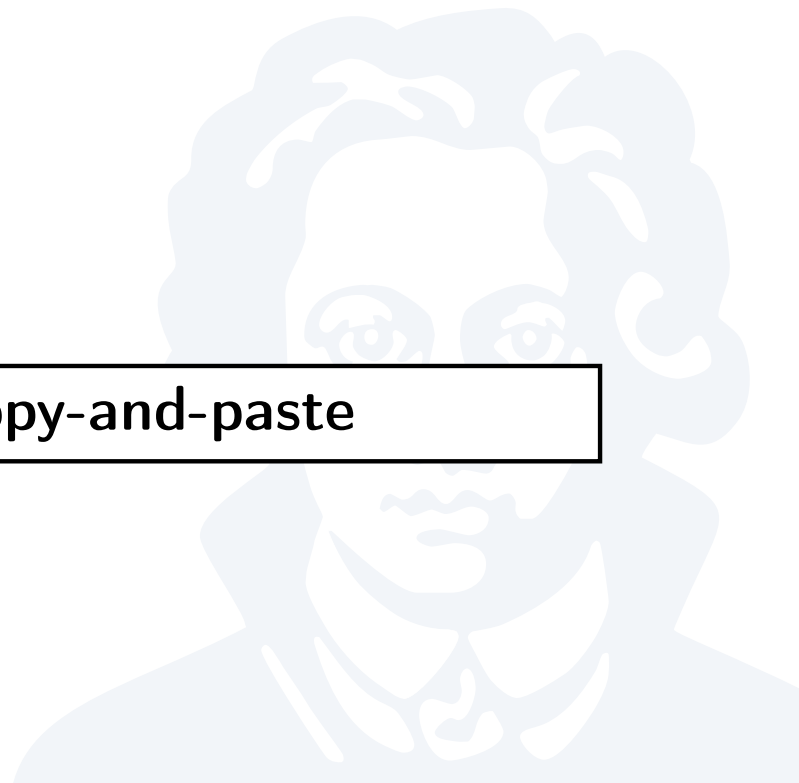
```
        li    $t0, 0x1
label:
        mul   $t0, $t1
        subi  $t1, 0x1
        bnez  $t1, label
```

Re-use through Subroutines

Let's recall our factorial code:

```
        li    $t0, 0x1
label:
        mul   $t0, $t1
        subi  $t1, 0x1
        bnez  $t1, label
```

Using subroutines avoids copy-and-paste



Calling and Returning

Call subroutine

`jals` ℓ

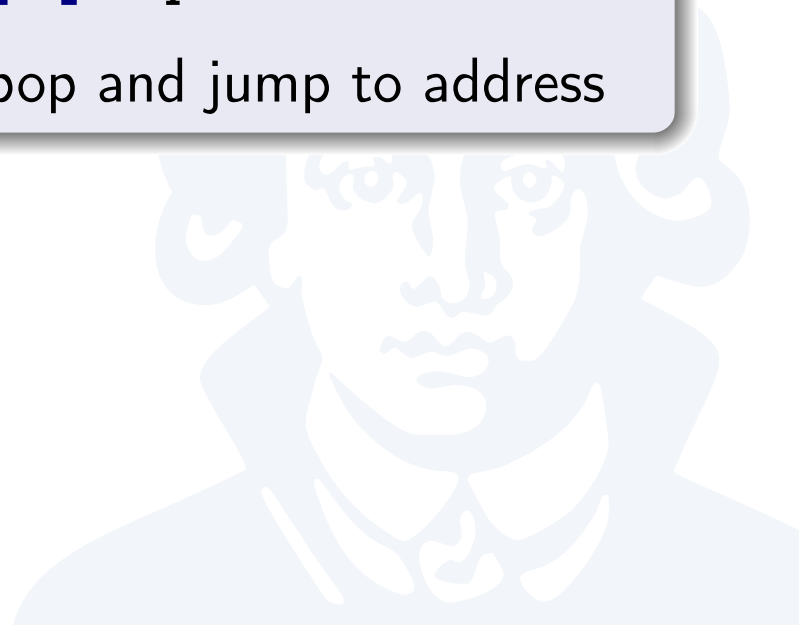
- **push** address of next instruction
- **j** ℓ

Return from subroutine

`jreturn`

- **pop** $pc

I.e., pop and jump to address



Subroutine invocation

Definition: factorial

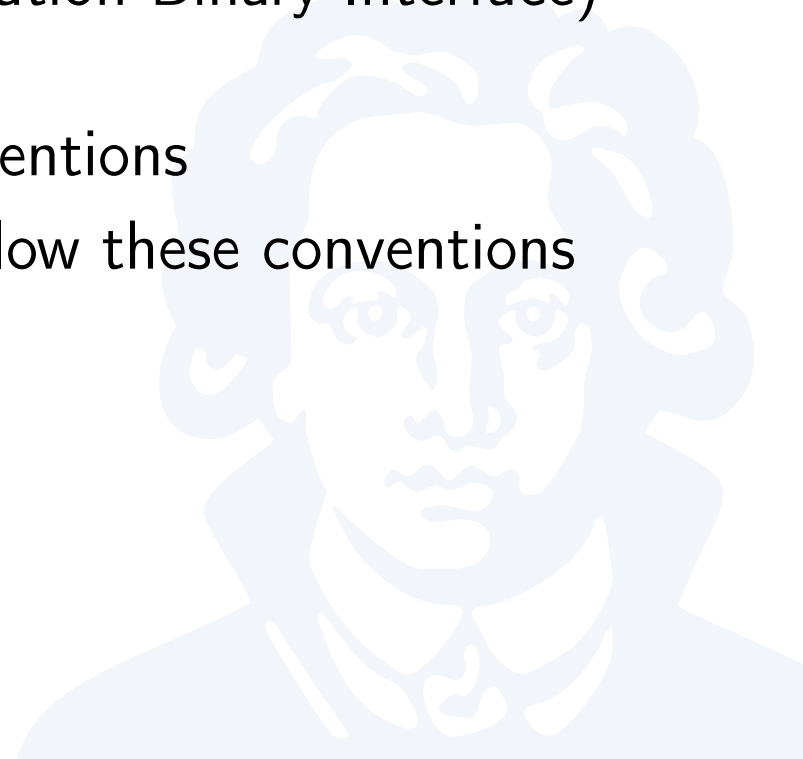
```
factorial:
            li      $v0, 0x1
loop:
            mul     $v0, $a0
            subi    $a0, 0x1
            bnez    $a0, loop
            jreturn
```

Invoking factorial

```
    li      $a0, 10         # Load parameter
    jals    factorial
            # Result in $v0
```

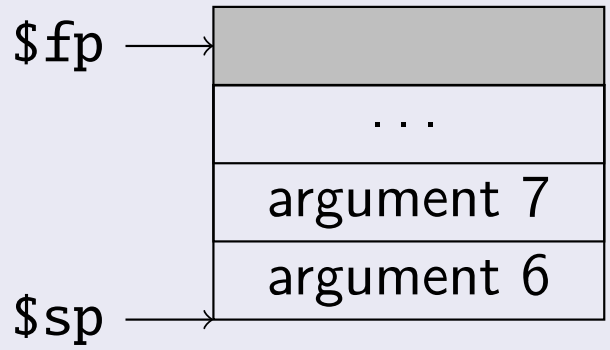
Using subroutines

- First parameter in `$a0`
- Return value in `$v0`
- Follows x86-64/Linux ABI (Application Binary Interface) *calling conventions*
- Existing libraries follow these conventions
- Existing libraries expect *you* to follow these conventions



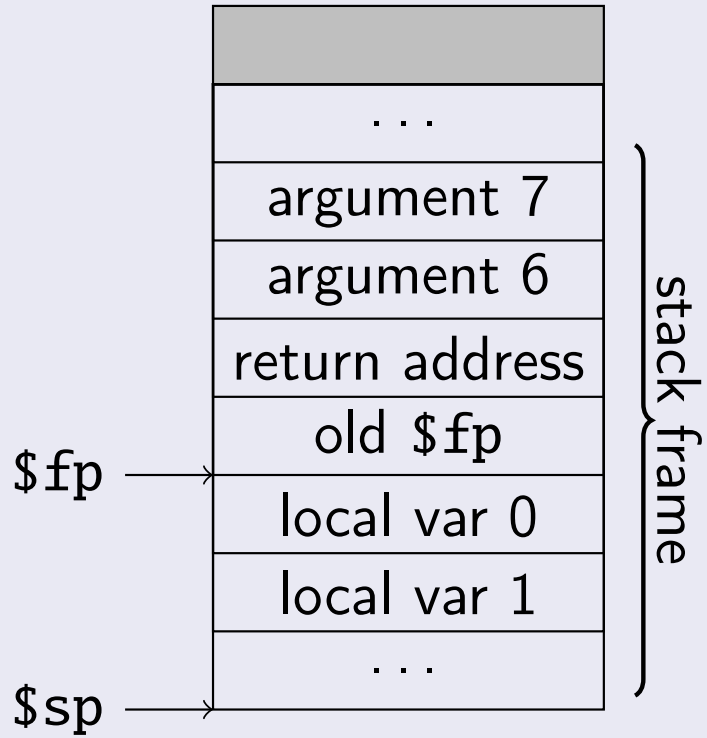
Stack and Invocations

Before Call



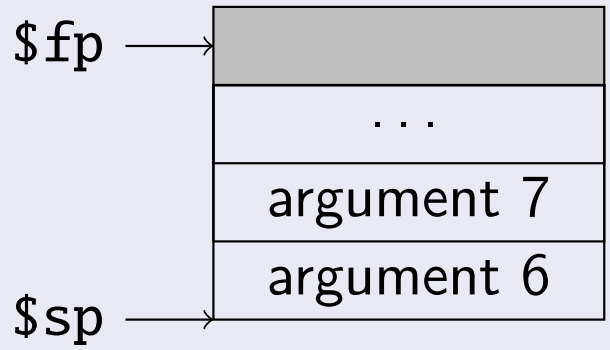
- $sp mod 16 = 8
- Arguments 0-5 in $a0–$a5

During Call



Stack and Invocations

Before Call

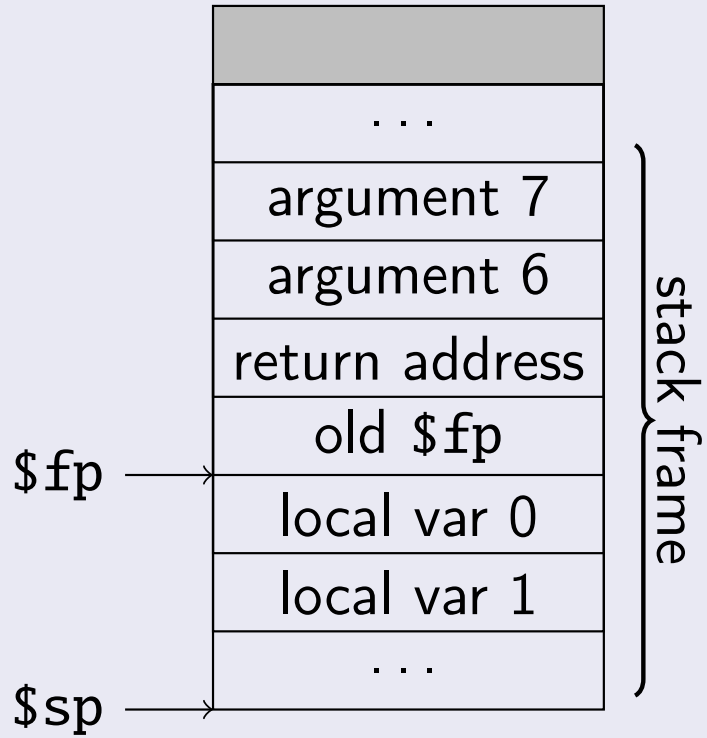


- $sp mod 16 = 8
- Arguments 0-5 in $a0–$a5

After Call

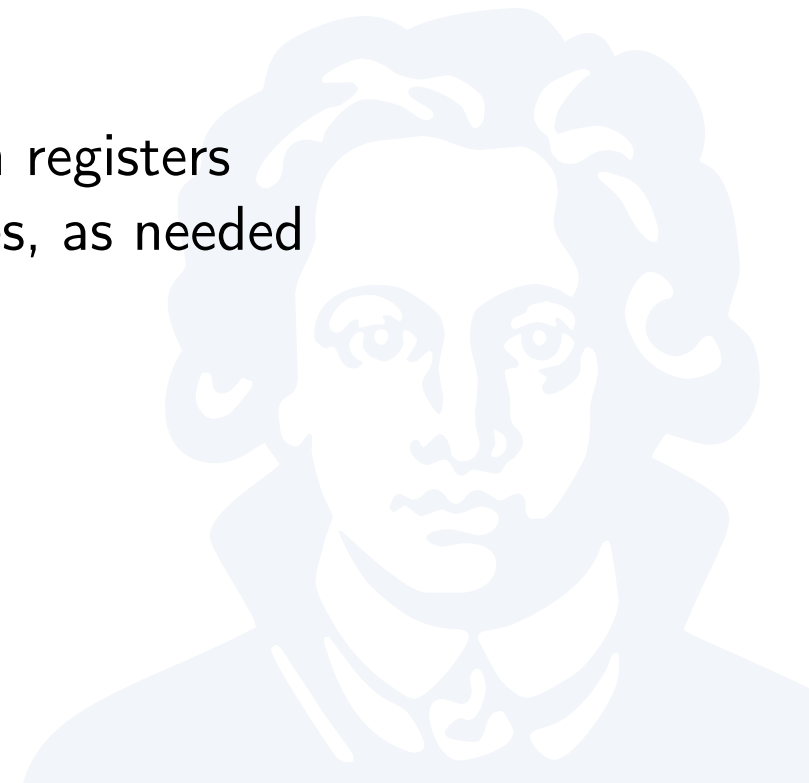
- $v0 contains result
- $sp, $fp, $gp, $s0–$s3 same as before call

During Call



Stack frame

- All data associated with a single function invocation: *Stack frame*
- Contains:
  - Return address
  - Parameters 6, 7, ...
  - Any local variables not stored in registers
  - Backups of callee-saved variables, as needed (`$fp`, `$s0`–`$s3`)
  - Other storage

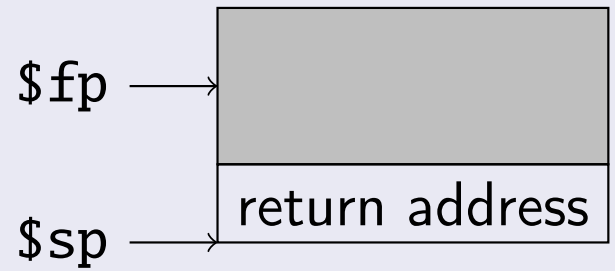


Recursive factorial

Definition: factorial

```
factorial:
            bnez    $a0, recurse
            li      $v0, 0x1
            jreturn
recurse:
            push    $fp
            move    $fp, $sp
            subi    $sp, 16
            sd      $a0, -8($fp)
            subi    $a0, 0x1
            jals    factorial
            ld      $a0, -8($fp)
            mul     $v0, $a0
            move    $sp, $fp
            pop     $fp
            jreturn
```

During Call

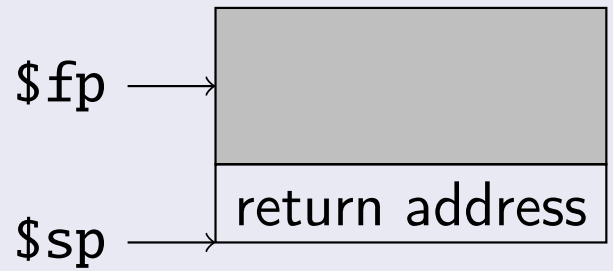


Recursive factorial

Definition: factorial

```
factorial:
⇒           bnez    $a0, recurse
⇒           li      $v0, 0x1
⇒           jreturn
recurse:
            push    $fp
            move    $fp, $sp
            subi    $sp, 16
            sd      $a0, -8($fp)
            subi    $a0, 0x1
            jals    factorial
            ld      $a0, -8($fp)
            mul     $v0, $a0
            move    $sp, $fp
            pop     $fp
            jreturn
```

During Call

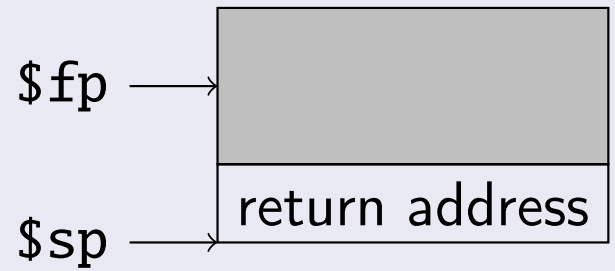


Recursive factorial

Definition: factorial

```
factorial:
            bnez    $a0, recurse
            li      $v0, 0x1
            jreturn
recurse:
⇒           push    $fp
            move    $fp, $sp
            subi    $sp, 16
            sd      $a0, -8($fp)
            subi    $a0, 0x1
            jals    factorial
            ld      $a0, -8($fp)
            mul     $v0, $a0
            move    $sp, $fp
            pop     $fp
            jreturn
```

During Call

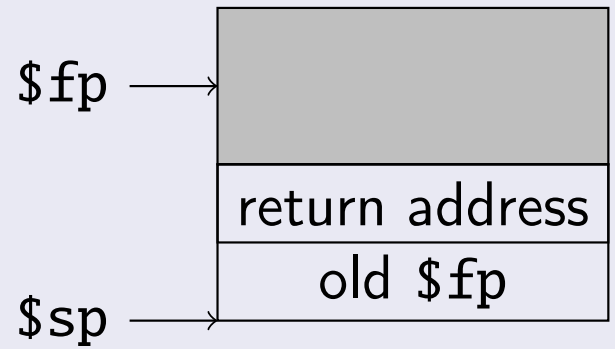


Recursive factorial

Definition: factorial

```
factorial:
                bnez    $a0, recurse
                li      $v0, 0x1
                jreturn
recurse:
                push    $fp
⇒               move    $fp, $sp
                subi    $sp, 16
                sd      $a0, -8($fp)
                subi    $a0, 0x1
                jals    factorial
                ld      $a0, -8($fp)
                mul     $v0, $a0
                move    $sp, $fp
                pop     $fp
                jreturn
```

During Call

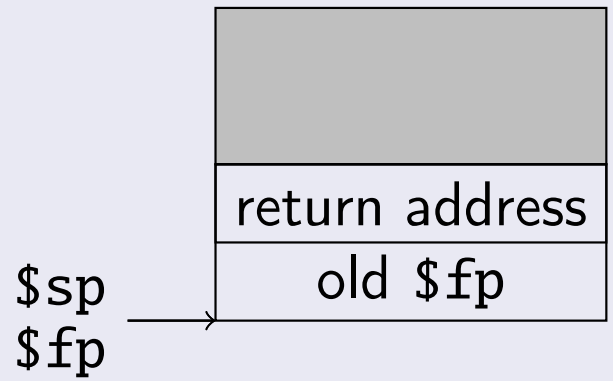


Recursive factorial

Definition: `factorial`

```
factorial:
            bnez    $a0, recurse
            li      $v0, 0x1
            jreturn
recurse:
            push    $fp
            move    $fp, $sp
⇒           subi    $sp, 16
            sd      $a0, -8($fp)
            subi    $a0, 0x1
            jals    factorial
            ld      $a0, -8($fp)
            mul     $v0, $a0
            move    $sp, $fp
            pop     $fp
            jreturn
```

During Call

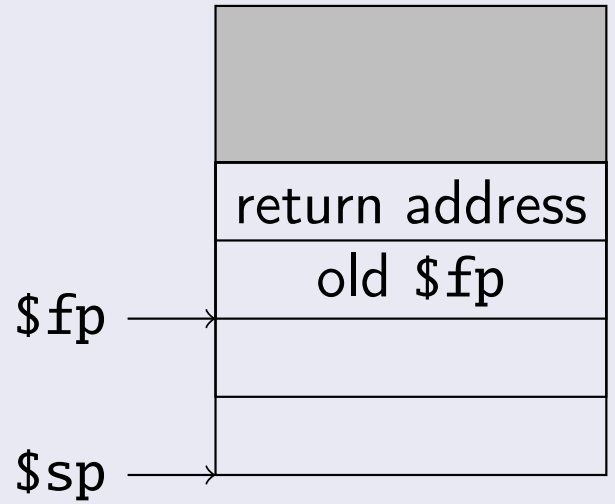


Recursive factorial

Definition: factorial

```
factorial:
                bnez    $a0, recurse
                li      $v0, 0x1
                jreturn
recurse:
                push    $fp
                move    $fp, $sp
                subi    $sp, 16
⇒               sd      $a0, -8($fp)
                subi    $a0, 0x1
                jals    factorial
                ld      $a0, -8($fp)
                mul     $v0, $a0
                move    $sp, $fp
                pop     $fp
                jreturn
```

During Call

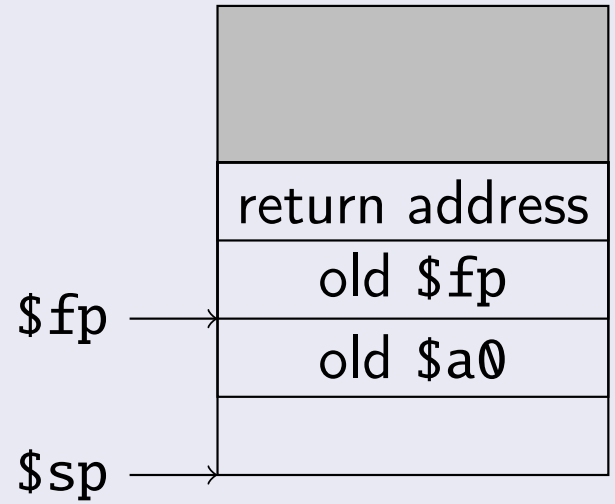


Recursive factorial

Definition: `factorial`

```
factorial:
            bnez    $a0, recurse
            li      $v0, 0x1
            jreturn
recurse:
            push    $fp
            move    $fp, $sp
            subi    $sp, 16
            sd      $a0, -8($fp)
⇒           subi    $a0, 0x1
            jals    factorial
            ld      $a0, -8($fp)
            mul     $v0, $a0
            move    $sp, $fp
            pop     $fp
            jreturn
```

During Call

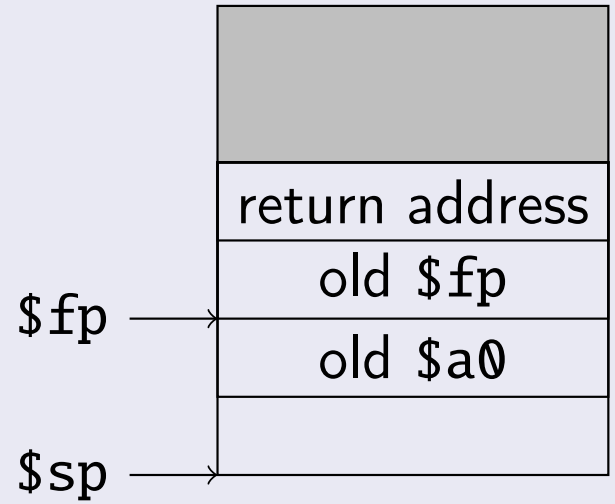


Recursive factorial

Definition: factorial

```
factorial:
            bnez    $a0, recurse
            li      $v0, 0x1
            jreturn
recurse:
            push    $fp
            move    $fp, $sp
            subi    $sp, 16
            sd      $a0, -8($fp)
            subi    $a0, 0x1
⇒           jals    factorial
            ld      $a0, -8($fp)
            mul     $v0, $a0
            move    $sp, $fp
            pop     $fp
            jreturn
```

During Call

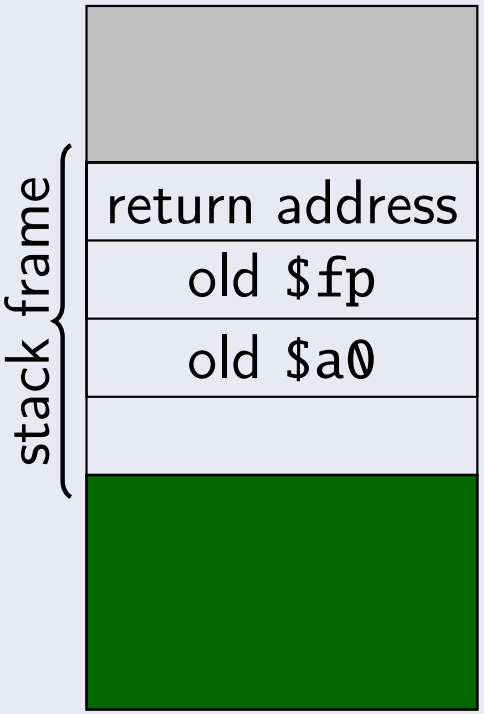


Recursive factorial

Definition: `factorial`

```
factorial:
            bnez    $a0, recurse
            li      $v0, 0x1
            jreturn
recurse:
            push    $fp
            move    $fp, $sp
            subi    $sp, 16
            sd      $a0, -8($fp)
            subi    $a0, 0x1
            jals    factorial
            ld      $a0, -8($fp)
            mul     $v0, $a0
            move    $sp, $fp
            pop     $fp
            jreturn
```

During Call

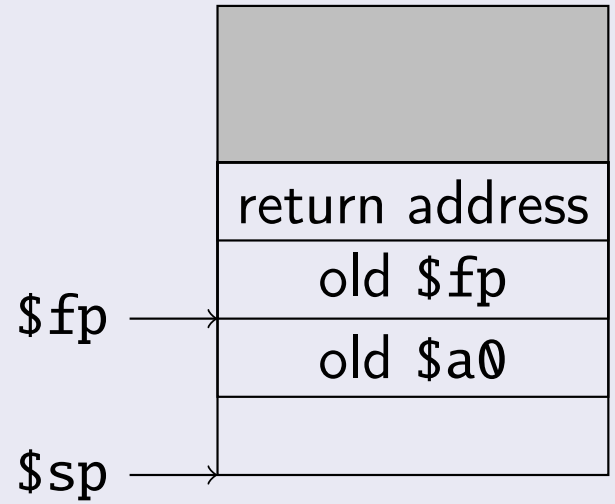


Recursive factorial

Definition: `factorial`

```
factorial:
            bnez    $a0, recurse
            li      $v0, 0x1
            jreturn
recurse:
            push    $fp
            move    $fp, $sp
            subi    $sp, 16
            sd      $a0, -8($fp)
            subi    $a0, 0x1
            jals    factorial
⇒           ld      $a0, -8($fp)
            mul     $v0, $a0
            move    $sp, $fp
            pop     $fp
            jreturn
```

During Call

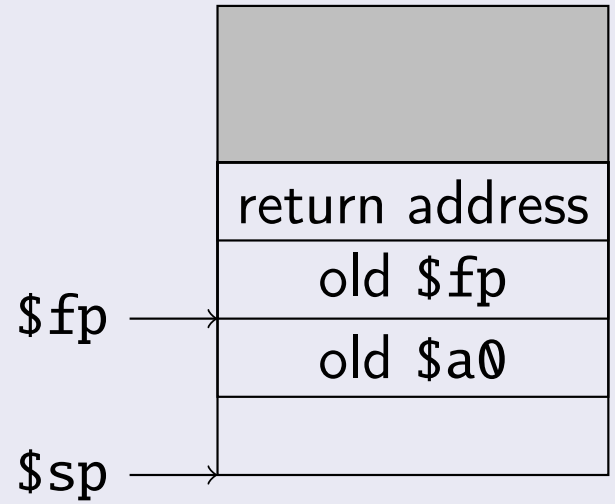


Recursive factorial

Definition: factorial

```
factorial:
            bnez    $a0, recurse
            li      $v0, 0x1
            jreturn
recurse:
            push    $fp
            move    $fp, $sp
            subi    $sp, 16
            sd      $a0, -8($fp)
            subi    $a0, 0x1
            jals    factorial
            ld      $a0, -8($fp)
⇒           mul     $v0, $a0
            move    $sp, $fp
            pop     $fp
            jreturn
```

During Call

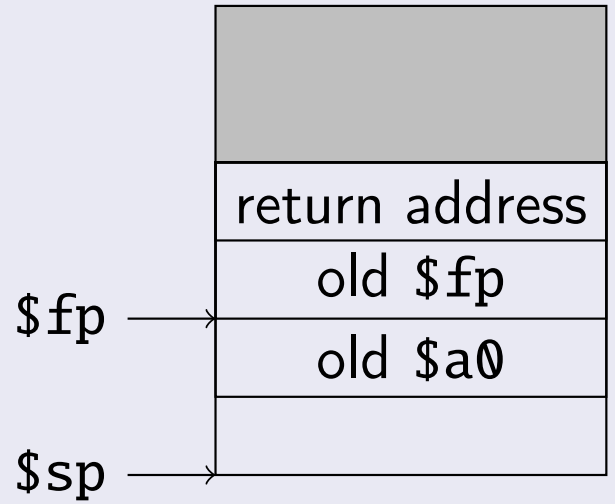


Recursive factorial

Definition: factorial

```
factorial:
            bnez    $a0, recurse
            li      $v0, 0x1
            jreturn
recurse:
            push    $fp
            move    $fp, $sp
            subi    $sp, 16
            sd      $a0, -8($fp)
            subi    $a0, 0x1
            jals    factorial
            ld      $a0, -8($fp)
            mul     $v0, $a0
⇒           move    $sp, $fp
            pop     $fp
            jreturn
```

During Call

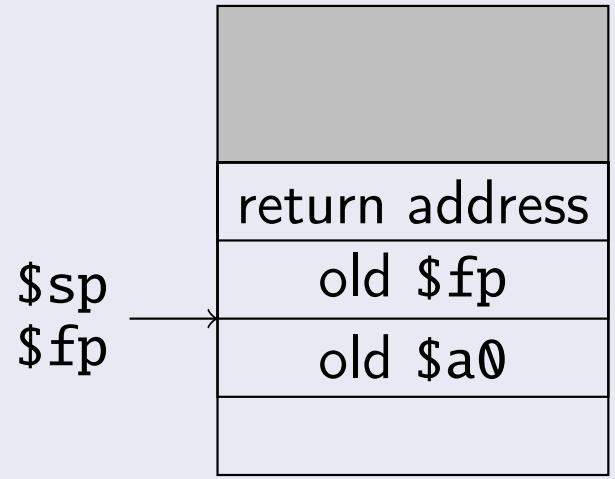


Recursive factorial

Definition: factorial

```
factorial:
            bnez    $a0, recurse
            li      $v0, 0x1
            jreturn
recurse:
            push    $fp
            move    $fp, $sp
            subi    $sp, 16
            sd      $a0, -8($fp)
            subi    $a0, 0x1
            jals    factorial
            ld      $a0, -8($fp)
            mul     $v0, $a0
            move    $sp, $fp
⇒           pop     $fp
            jreturn
```

During Call

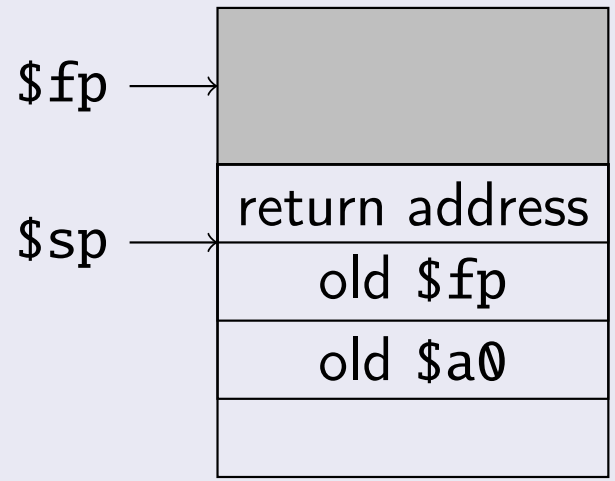


Recursive factorial

Definition: factorial

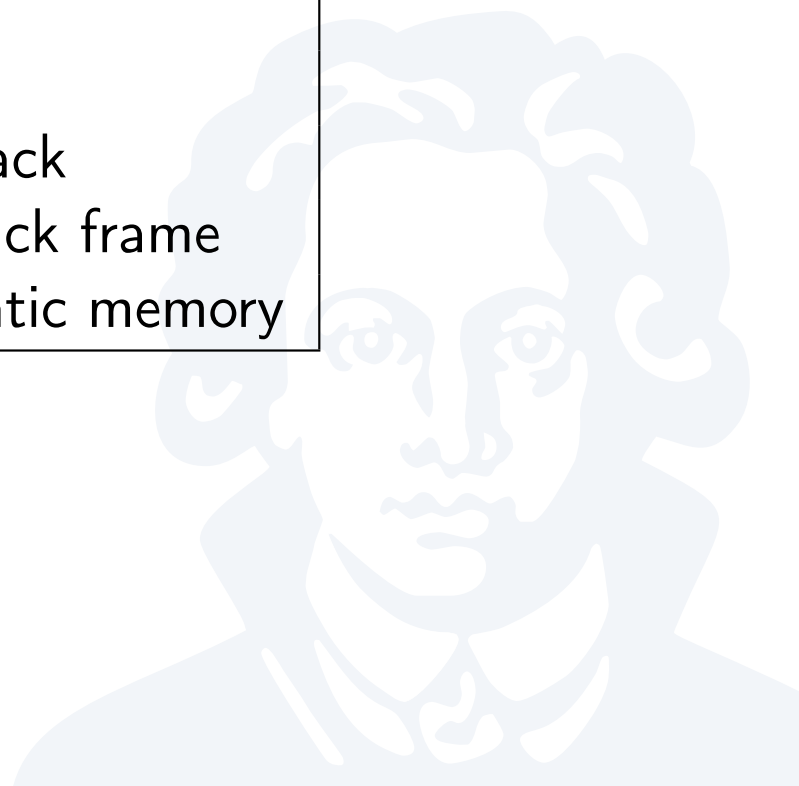
```
factorial:
            bnez    $a0, recurse
            li      $v0, 0x1
            jreturn
recurse:
            push    $fp
            move    $fp, $sp
            subi    $sp, 16
            sd      $a0, -8($fp)
            subi    $a0, 0x1
            jals    factorial
            ld      $a0, -8($fp)
            mul     $v0, $a0
            move    $sp, $fp
            pop     $fp
⇒           jreturn
```

During Call



2OPM Registers

Name	Purpose
`$v0`	Return **V**alue
`$a0`–`$a5`	**A**rguments
`$s0`–`$s3`	**S**aved registers: Callee-saved
`$t0`, `$t1`	**T**emporary registers
`$sp`	**S**tack **P**ointer: bottom of stack
`$fp`	**F**rame **P**ointer: access to stack frame
`$gp`	**G**lobal **P**ointer: access to static memory



Summary

- Subroutines allow re-using assembly instructions
- Means for implementing functions, methods, procedures, ...
- **`jals`**: call subroutine, store return address on stack
- **`jreturn`**: return from subroutine, load return address from stack
- Follow ABI conventions:
  - Routine arguments passed in `$a0`–`$a5` and stack
  - Return value in `$v0`
  - `$sp`, `$fp`, `$gp`, `$s0`–`$s3` preserved
  - All other registers may be altered
  - Function entry: `$sp` 16-byte aligned

Full list of operations in documentation